Think Like a Programmer

Practical Java solutions for students, developers, and interview preparation

Aaron Ploetz

bpb

Think Like a Programmer

Practical Java solutions for students, developers, and interview preparation

Aaron Ploetz

www.bpbonline.com

First Edition 2026

Copyright © BPB Publications, India

ISBN: 978-93-65899-054

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

*My wife **Coriene**, whose constant love and support keep me moving on to bigger and better things.*

About the Author

Aaron Ploetz is a developer advocate for IBM. He has been a professional software developer since 1997, and has a successful history of leading DBA and DevOps teams for startups and Fortune 50 enterprises. Aaron was named an MVP for Apache Cassandra® three times, a Cassandra Catalyst in 2023, and recently accepted an invitation to join the Apache Cassandra project as a committer. He has presented at many events, including Community over Code, Data Day Texas, and keynoted the fintech track at Dia Ciudad de México in 2023. Aaron has previously authored *Code with Java 21,* as well as books about distributed databases. He earned a B.S. in management/computer systems from the University of Wisconsin – Whitewater, and an M.S. in software engineering (database technology emphasis) from Regis University. When not writing or coding, Aaron enjoys fishing, retro video gaming, and skiing. Aaron and his wife, Coriene, live with their children in the Twin Cities area.

About the Reviewers

- **Dusan** is a senior software engineer and application security tech lead with over a decade of experience in software engineering. From the very start of his career, he was driven by a passion for continuous learning and using new technologies. This led him to become a full stack developer, allowing him to master both front-end and back-end technologies, including Java, Spring, Angular and Laravel, and to collaborate directly with designers and UX/UI experts.

  This excitement for learning ultimately drew him into the field of application security. He sees security as an ideal specialization because it intersects with various IT and computer science domains and demands constant curiosity.

  He has progressed through his career from a full stack developer to an application security expert, and finally to a security tech lead, where he now leads various application and cloud security initiatives.

- **Lesia Yanytska** is an accomplished product manager and technology strategist with extensive experience leading cross-industry digital transformation initiatives. With a strong foundation in business analysis and product ownership, she has successfully guided complex enterprise solutions from concept to scale across different domains. Lesia's work centers on building intelligent, user-centric products that bridge business goals with technical innovation. She excels at translating complex requirements into scalable architectures, fostering collaboration across global teams, and driving measurable business outcomes through data-driven decision-making.

- **Rafael** is a Brazilian professional, born in 1982 in Porto Alegre, Rio Grande do Sul. He has been living in Portugal since 2021 and holds a Resident Permit (Tech Visa) that allows him to live and work in the country.

His areas of expertise span multiple business domains, including banking and financial services, telecommunications, media and entertainment, healthcare, fashion, aviation, retail and e-commerce, gambling, transport, compliance, logistics, and fintech.

Acknowledgement

This book would not have come into being, were it not for the Java community. For decades, developers, technical writers, educators, and open-source committers around the world have shaped Java into this powerful craft that we all share. Their commitment to provide their knowledge not only makes Java better, but all of us as well.

Java continues to thrive because of this community and its commitment. It is my sincere hope that this book can serve the community similarly; as a useful resource to those on a journey to simply be better.

Preface

Welcome to *Think Like a Programmer!* This book is intended to help programmers develop and improve upon their logical reasoning and problem-solving abilities. In the pages that follow, we will discuss fundamental programming structures and approaches using the Java programming language. Each question will be broken down into a statement of the problem, a short description of the algorithm to be used, a visual approach using flowcharts, and (of course) the Java code to solve the problem.

The lessons shown in this book will be broken down into three sections based on complexity and subject matter. This will help readers to gain a solid understanding of the presented material, while being able to focus on consistent programming techniques and approaches. Additionally, each chapter and section will build upon the prior lessons learned, allowing readers to develop a solid programming foundation.

This book uses Java as its vehicle for teaching fundamental programming concepts. Therefore, some prior familiarity with Java is encouraged. To complete the lessons and exercises within, a current version of the **Java Development Kit** (**JDK**) and a Java-capable **integrated development environment** (**IDE**) are required. At the time of this book's publishing, the current release is Java 25, which is categorized as a **long-term support** (**LTS**) release. It is recommended for readers to build their local, development environment with a JDK of version 21 (the previous LTS release) or greater.

Readers should also have at least a beginner-level understanding of the Java language and syntax. While we may occasionally discuss some syntactical or functional nuances, this book will be more focused on solving the problems at-hand. Therefore, it is important that readers understand fundamental Java concepts like classes, objects, constructors, and access modifiers.

The choice of IDE to be used is entirely up to the reader. There are many great IDEs available that easily integrate with the Java ecosystem. IDEs like IntelliJ, Apache NetBeans, Eclipse, and VS Code are all great choices. The exercises for this book were built using the Eclipse IDE for Java developers, with Maven for dependency management.

The chapters will be grouped by section, and are described below:

Section 1: Beginner Problems

Chapter 1: Simple Loops — Starting simple, this chapter will show how to control program logic with different kinds of loops. We will also work with conditional statements and even use some simple arrays.

Chapter 2: File Operations — Here we will solve problems around reading and writing to files, as well as why to use the BufferedReader and BufferedWriter classes. We will also work with different types of files, including text, CSV, and even files requiring sensitive data to be hashed.

Chapter 3: Characters and Strings — This chapter will focus on problems involving processing and manipulating text data. We will also spend time learning how to appropriately use both primitive and reference data types.

Section 2: Intermediate Problems

Chapter 4: Arithmetic Solutions — In this chapter we will learn to solve problems with basic arithmetic and even build some of our own operation methods to use. Along the way, we will also discuss recursion, and work with numbers in a base 2 or binary format as well.

Chapter 5: Basic Data Structures — While this chapter has a smaller number of programs, the problems it solves are those which are fundamental to understanding programming logic. Here we will cover stacks, queues, and other structures which are ubiquitous in their use throughout computing.

Chapter 6: Pattern Matching — In this chapter, we will focus more on ways to match string patterns and object types. We will demonstrate more ways to leverage regular expressions and have a look at the pattern matching feature with Java's switch/case

statement.

Section 3: Challenging Problems

Chapter 7: Complex Data Structures — Here we will begin with a familiar data structure (a linked list) and then progress into more challenging structures like heaps. Each structure will also be shown in the context of solving real-world problems.

Chapter 8: Sorting — This chapter will focus on different ways to sort data sets. We will also discuss a little about algorithm efficiency, specifically with time/space complexity or Big O notation.

Chapter 9: Graphics — For the last chapter, we will work through some simple exercises around displaying colorful graphics and animations. Part of our animation work will lead us to work a little with virtual threads, as well.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/ff9e04

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Think-Like-a-Programmer**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks. You can check our social media handles below:

Get in touch with us at: **business@bpbonline.com** for more details.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Table of Contents

Section 1: Beginner Problems

1. Simple Loops

Introduction

Structure

Objectives

Program 1.1: Print the contents of an array

Algorithm

Flowchart

Java code

Code explanation

Program 1.2: Reverse and print an array

Algorithm

Flowchart

Java code

Code explanation

Program 1.3: FizzBuzz

Algorithm

Flowchart

Java code

Code explanation

Program 1.4: Print a diamond of numbers

Algorithm

Flowchart

Java code

Code explanation
Program 1.5: Diamond outline of numbers
Algorithm
Flowchart
Java code
Code explanation
Program 1.6: Checking valid number using while
Algorithm
Flowchart
Java code
Code explanation
Program 1.7: Display user names
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

2. File Operations

Introduction
Structure
Objectives
Program 2.1: Write to a text file
Algorithm
Flowchart
Java code
Code explanation
Program 2.2: Read and print from a text file
Algorithm
Flowchart
Java code

Code explanation
Program 2.3: Read and print from a CSV file
Algorithm
Flowchart
Java code
Code explanation
Program 2.4: Write a password file
Algorithm
Flowchart
Java code
Code explanation
Program 2.5: Validate a user from a password file
Algorithm
Flowchart
Java code
Code explanation
Program 2.6: Write a unique password file
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

3. Characters and Strings

Introduction
Structure
Objectives
Exploring instance and static methods
Program 3.1: Reverse a string
Algorithm
Flowchart

Java code
Code explanation
Program 3.2: Check if a primitive character is uppercase
Algorithm
Flowchart
Java code
Code explanation
Program 3.3: Check if a reference character is uppercase
Algorithm
Flowchart
Java code
Code explanation
Program 3.4: Format a phone number
Algorithm
Flowchart
Java code
Code explanation
Program 3.5: Check if a state code is valid
Algorithm
Flowchart
Java code
Code explanation
Program 3.6: Check if a state code is valid using regular expression
Algorithm
Flowchart
Java code
Code explanation
Program 3.7: Check if a string is a palindrome
Algorithm
Flowchart
Java code
Code explanation

Program 3.8: Parse and format hockey team data
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

Section 2: Intermediate Problems

4. Arithmetic Solutions
Introduction
Structure
Objectives
Program 4.1: Multiply two numbers
Algorithm
Flowchart
Java code
Code explanation
Program 4.2: Subtract two numbers
Algorithm
Flowchart
Java code
Code explanation
Program 4.3: Divide two numbers
Algorithm
Flowchart
Java code
Code explanation
Program 4.4: Check if a number is a power of two
Algorithm
Flowchart

Java code
Code explanation
Program 4.5: Build a binary error handler
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

5. Basic Data Structures
Introduction
Structure
Objectives
Program 5.1: Build a stack to process truck pallets
Algorithm
Flowchart
Java code
Code explanation
Program 5.2: Build a queue for airport takeoffs
Algorithm
Flowchart
Java code
Code explanation
Program 5.3: Build a linked list for navigation
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

6. Pattern Matching

Introduction

Structure

Objectives

Program 6.1: Determine a valid phone number

Algorithm

Flowchart

Java code

Code explanation

Program 6.2: Determine if a new password is valid

Algorithm

Flowchart

Java code

Code explanation

Program 6.3: Determine if an IP address is valid

Algorithm

Flowchart

Java code

Code explanation

Program 6.4: Compute tollway charges by vehicle

Algorithm

Flowchart

Java code

Code explanation

Conclusion

Points to remember

Questions

Section 3: Challenging Problems

7. Complex Data Structures

Introduction

Structure

Objectives
Program 7.1: Build a linked list for browser history
Algorithm
Flowchart
Java code
Code explanation
Program 7.2: Build a binary tree to store fruit data
Algorithm
Flowchart
Java code
Code explanation
Program 7.3: Build a heap for a medical queue
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

8. Sorting

Introduction
Structure
Objectives
Program 8.1: Sort users with a bubble sort
Algorithm
Flowchart
Java code
Code explanation
Program 8.2: Sort users with a merge sort
Algorithm
Flowchart
Java code

Code explanation
Program 8.3: Sort with a binary tree sort
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

9. Graphics
Introduction
Structure
Objectives
Program 9.1: Draw the moon with a starfield
Algorithm
Flowchart
Java code
Code explanation
Program 9.2: Animate a bouncing ball
Algorithm
Flowchart
Java code
Code explanation
Program 9.3: Animate multiple bouncing balls
Algorithm
Flowchart
Java code
Code explanation
Conclusion
Points to remember
Questions

APPENDIX A: References

APPENDIX B: Answers to Exercises

- Chapter 1
- Chapter 2
- Chapter 3
- Chapter 4
- Chapter 5
- Chapter 6
- Chapter 7
- Chapter 8
- Chapter 9

Index

Section 1:
Beginner Problems

CHAPTER 1

Simple Loops

Introduction

This chapter's main focus will be to introduce the concept of simple looping constructs used in Java. We will cover how to implement logic using for, while, and do-while loops.

We will also introduce flowcharts in this chapter. A flowchart is a visual representation of how logic progresses inside a computer program or data structure. Inside a flowchart, specific components are used to convey inputs, decisions, and processes. Some of the basic flowchart components can be seen in the following *Figure 1.1*:

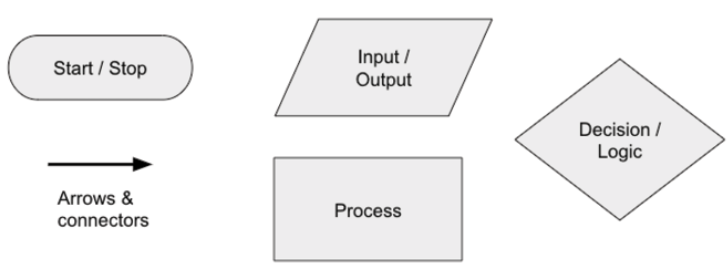

***Figure 1.1**: Basic flowchart components to be used in this book*

Here is a description of how we will use each of these symbols:

- **Start/stop**: Indicates the beginning or end of a program or logic flow.
- **Arrows and connectors**: These represent the flow of data or logic within a program and its components.
- **Input/output**: Used to indicate when data is input into a program, or when it is returned from a program.
- **Process**: This component is used to indicate simple, compartmentalized operations within a program.
- **Decision/logic**: Used to show different logic paths through the program.

As the book progresses and the complexity of the problems to be solved increases, we may introduce additional components. They will be described and used on an as-needed basis.

Structure

In this chapter, we will cover the following:

- Program 1.1: Print the contents of an array
- Program 1.2: Reverse and print an array
- Program 1.3: FizzBuzz
- Program 1.4: Print a diamond of numbers
- Program 1.5: Diamond outline of numbers
- Program 1.6: Checking valid number using while
- Program 1.7: Display user names

Objectives

In this chapter, we will examine some simple programming problems as a way to build a foundation for further chapters. We will also introduce flowcharts as a tool to visually represent how we go about solving the problems. Our programming problems will help to provide a basic understanding of looping, arrays, conditional

statements, nested loops, and conditional techniques and constructs.

Program 1.1: Print the contents of an array

Take a simple list (array) of numbers and print them to the standard output.

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **printArray**, which accepts an array of integers named **array**.
2. Create a for-each loop to iterate through each number in the **array**.
3. Print the current value of the **number** to the screen.
4. Print an additional carriage return.
5. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 1.2*:

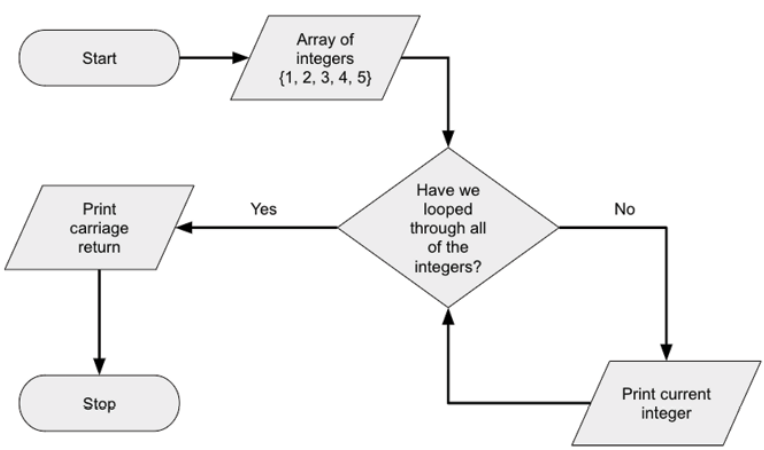

***Figure 1.2**: Flowchart describing how to print all numbers in an array*

Java code

Here is the Java code for *Program 1.1*:

```
private static void printArray(Integer[] array) {
    for (Integer number: array) {
        System.out.println(number);
    }
    System.out.println();
}
```

Code explanation

To solve this problem, we will create a new method named **printArray**, and it will accept an array of integers as a parameter. We will use a *for-each* loop to iterate through and print each number in the **array**. Once the loop is complete, we will print an extra carriage return to provide some level of spacing after the printed output.

Note: This program could also be written using the primitive type int instead of the reference type Integer. We will use Integer here because it makes it easier to print each number. The differences between reference and primitive types will be discussed in a later chapter.

Program 1.2: Reverse and print an array

Take a simple list (array) of numbers and print them in reverse order.

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **reverseArray**, which accepts an array of integers named **array** as a parameter.
2. Initialize a new, empty array of integers (matching the size of the array) named **reversedArray** to hold the reversed array values.
3. Create an index for the reversed array named **reversedIndex**, and initialize it to 0.

4. Create a for loop with a counter named **arrayIndex**. Initialize **arrayIndex** to the size of the array, and decrement through the array from back to front.
5. Take the number at the current array position of (**arrayIndex** minus 1) and insert it into **reversedArray** at the position designated by the value of **reversedIndex**.
6. Increment **reversedIndex**.
7. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 1.3*:

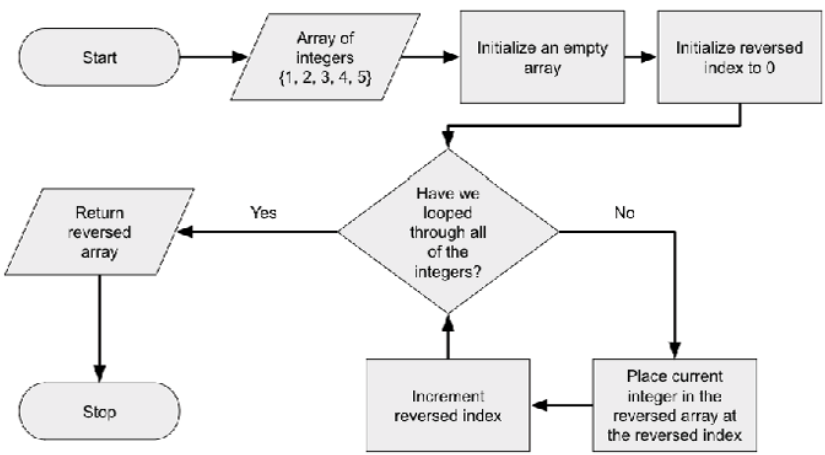
Start
Array of integers {1, 2, 3, 4, 5}
Initialize an empty array
Initialize reversed index to 0
Have we looped through all of the integers?
Yes
No
Return reversed array
Stop
Increment reversed index
Place current integer in the reversed array at the reversed index

***Figure 1.3**: Flowchart showing how to reverse the numbers in an array*

Java code

Here is the Java code for *Program 1.2:*

```
private static Integer[] reverseArray(Integer[] array) {
    Integer[] reversedArray = new Integer[array.length];
    int reversedIndex = 0;
    for (int index = array.length; index > 0; index--) {
        reversedArray[reversedIndex] = array[index - 1];
```

```
        reversedIndex++;
    }
    return reversedArray;
}
```

Code explanation

First, we create a method named **reverseArray** that both accepts and returns an array of integers. Inside that method, we begin by instantiating variables to hold values for our reversed array and our reversed index (named **reversedArray** and **reversedIndex**, respectively).

We then iterate through all the numbers in the input array with a for loop. The only difference from the last program is that we will start from the end of the **array** and work our way through it backwards. As we decrement our way through the **array**, we will add the current number to **reversedArray**, using the position designated by **reversedIndex**.

Once the for loop has iterated through all the available numbers in the **array**, it will return the **reversedArray** to the calling program. In our calling program, we will call the **printArray()** method (from *Program 1.1*) to print the contents of the reversed array to the console.

Program 1.3: FizzBuzz

Loop through the numbers 1 to 20. If a number is divisible by 3, print **Fizz**. If the number is divisible by 5, print **Buzz**. If the number is divisible by both 3 and 5, print **FizzBuzz**.

Note: This problem gained some notoriety for being used as an interview question at Stack Overflow.

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **fizzBuzz** which accepts an integer named **maxCount** as its parameter.

2. Create a for loop with a counter named **counter**, and initialize it to a value of 1. Iterate as long as counter is less than or equal to the value of **maxCount** (20 in this example).
3. Check if the current value of **counter** is evenly divisible by both 3 and 5 (**counter % 3 == 0 && counter % 5 == 0**). If so, print the value of the **counter** and the string **FizzBuzz**.
4. Check if the current value of the **counter** is evenly divisible only by 3 (**counter % 3 == 0**). If so, print the value of the **counter** and the string **Fizz**.
5. Check if the current value of the **counter** is evenly divisible only by 5 (**counter % 5 == 0**). If so, print the value of the **counter** and the string **Buzz**.
6. Print a carriage return.
7. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 1.4*:

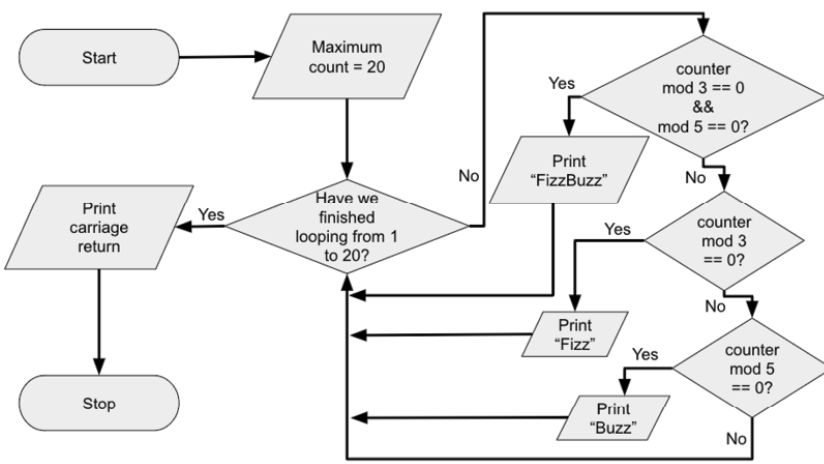

***Figure 1.4**: Flowchart showing how to solve the FizzBuzz problem*

Java code

Here is the Java code for *Program 1.3:*

```
private static void fizzBuzz(int maxCount) {
    for (int counter = 1; counter <= maxCount; counter++) {
        if (counter % 3 == 0 && counter % 5 == 0) {
            System.out.print("FizzBuzz = ");
            System.out.println(counter);
        } else if (counter % 3 == 0) {
            System.out.print("Fizz = ");
            System.out.println(counter);
        } else if (counter % 5 == 0) {
            System.out.print("Buzz = ");
            System.out.println(counter);
         }
    }
    System.out.println();
}
```

Code explanation

First, we create a method named **fizzBuzz** that accepts a maximum number for us to count to. We will initialize a for loop and increment the loop's counter variable by 1 for each iteration.

Inside the loop, we will make three evaluations using the % symbol, which is Java's modulus operator. If the result of the **counter**'s modulus of 3 is 0, that means it is evenly divisible by 3. Likewise, if the result of the **counter**'s modulus of 5 is 0, that means it is evenly divisible by 5. However, we will want to check for the condition that the **counter** is divisible by both 3 and 5 first; otherwise, the logic will short-circuit if it is divisible by one or the other.

In any case, if the **counter** is divisible by both 3 and 5, we will print the value of the **counter** with the string **FizzBuzz**. If it is divisible only by 3, we will print the value of the **counter** with the string **Fizz**. Finally, if only divisible by 5, we will print the value of the **counter** with the string **Buzz**.

Once the loop is complete, we will print an additional carriage

return for spacing.

Note: Printing the value of the counter is not a requirement for solving the problem. We are doing it here to help ensure the understanding of the logic.

Program 1.4: Print a diamond of numbers

Given a single-digit number, print a diamond shape of numbers. The number entered should be the middle point of the diamond, with all others ascending or descending around it. The numbers at the top of the diamond should increase until they reach the number entered, and they should decrease at the bottom of the diamond. When complete, the diamond should look like this (when seeded with a number value of 5):

```
    1
   222
  33333
 4444444
555555555
 4444444
  33333
   222
    1
```

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **printDiamond** that accepts an integer named **size**.
2. If the value of **size** is >= 10, then:
    a. Print **Please provide a single-digit number**.
    b. Return to the calling program.
3. Create a for loop with a counter named **topCounter**. Initialize **topCounter** to 1 and increment as long as **topCounter** is less than or equal to the value of size.
    a. Create a new for loop nested inside the **topCounter** loop, to print the leading spaces with a counter named

spaceCounter. Initialize **spaceCounter** to 0 and increment as long as **spaceCounter** is less than the result of **size** minus the value of **topCounter**. Inside the loop, it will print a single space.

b. Create a new for loop (after the **spaceCounter** loop, but inside of the **topCounter** loop). Initialize the loop with a counter named **numCounter** with a value of 0. Increment **numCounter** while iterating through the loop, as long as the value of **numCounter** is less than **topCounter** multiplied by 2, minus 1. Inside the loop, it will print the current value of **topCounter**.

c. Print a carriage return.

4. Create a for loop with a counter named **bottomCounter**. Initialize **bottomCounter** to a value of size minus 1, and decrement as long as **bottomCounter** is greater than 0.

   a. Create a new for loop nested inside the previous for loop, to print the leading spaces with a counter named **spaceCounter2**. Initialize **spaceCounter2** to 0 and increment it as long as **spaceCounter2** is less than the result of **size** minus the value of **bottomCounter**. Inside the loop, it will print a single space.

   b. Create a new for loop (after the **spaceCounter2** loop, but inside of the **bottomCounter** loop). Initialize the loop with a counter named **numCounter2** with a value of 0. Increment **numCounter2** while iterating through the loop, as long as the value of **numCounter2** is less than **bottomCounter** multiplied by 2, minus 1. Inside the loop, it will print the current value of **bottomCounter**.

   c. Print a carriage return.

Flowchart

The flowchart for this problem can be seen in *Figure 1.5*:

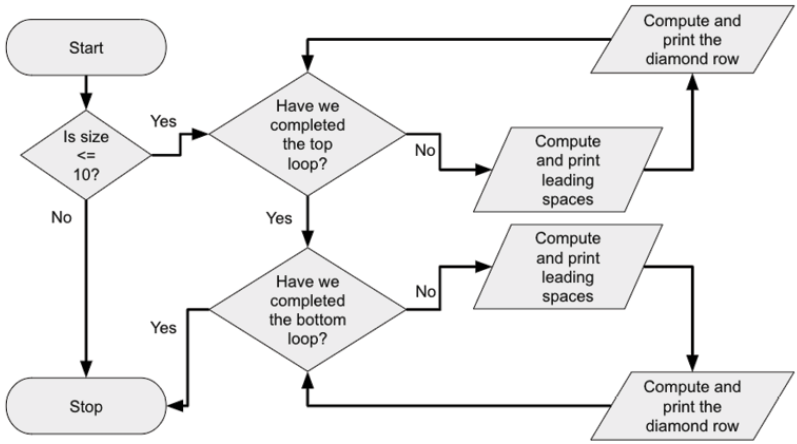

***Figure 1.5**: Flowchart showing how to solve the diamond of numbers problem*

Java code

The following is the Java code for *Program 1.4:*

```
private static void printDiamond(int size) {
    if (size < 10) {
        // upper half
        for (int topCounter = 1; topCounter <= size; topCounter++)
{
            // compute and print leading spaces
            for (int spaceCounter = 0; spaceCounter < (size -
topCounter);
                    spaceCounter++) {
                System.out.print(" ");
            }
            // print number(s)
            for (int numCounter = 0; numCounter < ((topCounter * 2) -
1);
                    numCounter++) {
                System.out.print(topCounter);
            }
            System.out.println();
```

```
        }
        // bottom half
        for (int bottomCounter = (size - 1); bottomCounter > 0;
bottomCounter--) {
            // compute and print leading spaces
            for (int spaceCounter2 = 0; spaceCounter2 < (size -
bottomCounter);
                    spaceCounter2++) {
                System.out.print(" ");
            }
            // print number(s)
            for (int numCounter2 = 0; numCounter2 <
((bottomCounter * 2) - 1);
                numCounter2++) {
                System.out.print(bottomCounter);
            }
            System.out.println();
        }
    } else {
        System.out.println("Please provide a single-digit number.");
    }
    System.out.println();
}
```

Code explanation

First, we create a new method named **printDiamond** that accepts an integer named **size** as its parameter. We check to ensure that **size** is less than 10 before proceeding any further. For this algorithm, double-digit numbers will complicate the required spacing, so we will avoid them for now. If we determine that **size** has a value of 10 or greater, we will print the string **Please provide a single-digit number**. and return to the calling program.

Inside the **if** statement previously, we will create a new for loop

with a counter named **topCounter**. We will initialize **topCounter** to a value of 1. Normally, we would initialize a loop counter to 0, but in this case, we do not want to work with the number 0; only the positive integers less than or equal to size (5 in this example). We will increment **topCounter** and iterate through the loop as long as **topCounter** is less than or equal to **size**.

Next, we will compute and print the leading spaces for the top section of the diamond. We can do this with a simple for loop (created or nested inside of the **topCounter** loop) that initializes a counter named **spaceCounter** to 0. We will increment **spaceCounter** and iterate through the loop (printing a single space) as long as **spaceCounter** is less than the value of size minus **topCounter**. This will generate a number of spaces that is inversely proportional to the value of **topCounter**.

We will then implement another for loop to print the numbers for each row of the diamond's top half. This loop will also be nested inside of the **topCounter** loop and will have a counter named **numCounter** that is initialized to a value of 0. This loop will simply print the value of **topCounter** (without a carriage return) as long as **numCounter** is less than the value of **topCounter** times 2, minus 1. This will print the diamond row to an appropriate length. Once this loop is complete, we will print a single carriage return.

With the top half of the diamond printed, we will create a for loop to bring the bottom half. It will have a counter named **bottomCounter** and initialized to a value of size minus 1. We will use size minus 1, because we do not want to print the middle row twice. We will decrement **bottomCounter** and iterate through the loop as long as **bottomCounter** is greater than a value of 0.

Similar to the top half of the diamond, we will also implement another for loop to print the numbers for each row of the diamond's bottom half. This loop will also be nested inside of the **bottomCounter** loop and will have a counter named **numCounter2** that is initialized to a value of 0. This loop will simply print the value of **bottomCounter** (without a carriage return) as long as **numCounter2** is less than the value of **bottomCounter** times 2, minus 1. Once this loop is complete, we will once again print a single carriage return.

When this is all complete, we should have our printed diamond as shown in the beginning of this problem.

Program 1.5: Diamond outline of numbers

This problem is similar to the one previously, except we will just print the outline of the diamond. Given a single-digit number, print a diamond shape outlined in numbers. The number entered should be the middle point of the diamond. The numbers at the top of the outline should increase until they reach the number entered, and they should decrease in the bottom of the outline. When complete, the diamond should look like this (when seeded with a number value of 5):

```
    1
   2 2
  3   3
 4     4
5       5
 4     4
  3   3
   2 2
    1
```

Algorithm

To run this program, we will follow these steps:

1. Create a new method named **printDiamondOutline** that accepts an integer named **size**.
2. If the value of **size** is > 10, then:
    a. Print **Please provide a single-digit number**.
    b. Return to the calling program.
3. Create a for loop with a counter named **topCounter**. Initialize **topCounter** to 1 and increment as long as **topCounter** is less than or equal to the value of size.
    a. Create a new for loop nested inside the previous for loop, to print the leading spaces with a counter named **spaceCounter**. Initialize **spaceCounter** to 0 and increment as long as **spaceCounter** is less than the

result of size minus the value of **topCounter**. Inside the loop, it will print a single space.

b. Check if **topCounter** is greater than 1. If it is:

i. Print the value of **topCounter**.

ii. Create a new for loop nested inside the if statement with a counter named **fillCounter** initialized to 0. Continue as long as **fillCounter** is less than **topCounter** multiplied by 2, minus 3. This will ensure appropriate spacing at the top of the diamond. Inside the loop, it will print a single space.

c. Print the value of **topCounter**.

4. Create a for loop with a counter named **bottomCounter**. Initialize **bottomCounter** to a value of **size** minus 1, and decrement as long as **bottomCounter** is greater than 0.

a. Create a new for loop nested inside the previous for loop, to print the leading spaces with a counter named **spaceCounter2**. Initialize **spaceCounter2** to 0 and increment it as long as **spaceCounter2** is less than the result of size minus the value of **bottomCounter**. Inside the loop, it will print a single space.

b. Check if **bottomCounter** is greater than 1. If it is, then it will print the value of **bottomCounter**.

i. Create a new for loop nested inside the if statement with a counter named **fillCounter2** initialized to 0. Continue as long as **fillCounter2** is less than **bottomCounter** multiplied by 2, minus 3. This will ensure appropriate spacing at the bottom of the diamond. Inside the loop, it will print a single space.

c. Print the value of **topCounter**.

Flowchart

The flowchart for this problem can be seen in *Figure 1.6*:

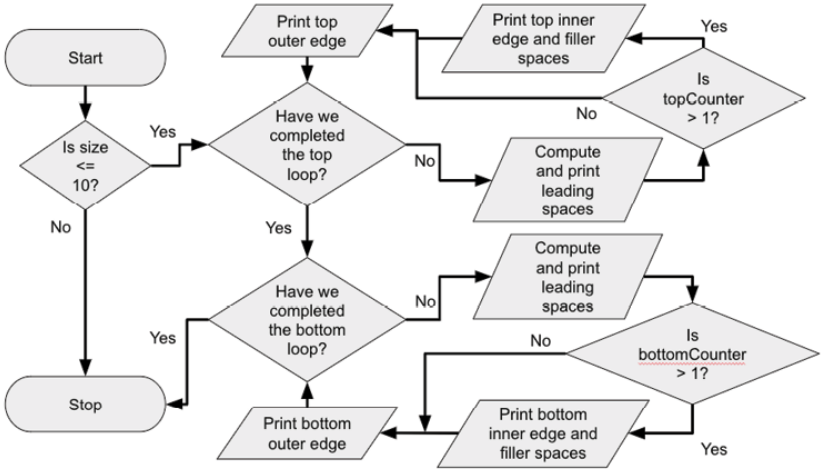

***Figure 1.6**: Flowchart showing how to solve the diamond outline problem*

Java code

Here is the Java code for *Program 1.5:*

```
private static void printDiamondOutline(int size) {
    if (size < 10) {
        // upper half
        for (int topCounter = 1; topCounter <= size; topCounter++)
{
            // compute and print leading spaces
            for (int spaceCounter = 0; spaceCounter < (size -
topCounter);
                    spaceCounter++) {
                System.out.print(" ");
            }
            // print number(s)
            if (topCounter > 1) {
                System.out.print(topCounter);
                // filler spaces, not needed for the first iteration
                for (int fillCounter = 0; fillCounter < ((topCounter * 2) -
3);
```

```
                fillCounter++) {
                System.out.print(" ");
            }
        }
        System.out.println(topCounter);
    }
    // bottom half
    for (int bottomCounter = (size - 1); bottomCounter > 0;
        bottomCounter--) {
        // compute and print leading spaces
        for (int spaceCounter2 = 0; spaceCounter2 < (size -
bottomCounter);
                spaceCounter2++) {
            System.out.print(" ");
        }
        // print number(s)
        if (bottomCounter > 1) {
            System.out.print(bottomCounter);
            // filler spaces, not needed for the first iteration
            for (int fillCounter2 = 0;
                    fillCounter2 < ((bottomCounter * 2) - 3);
                    fillCounter2++) {
                System.out.print(" ");
            }
        }
        System.out.println(bottomCounter);
    }
  } else {
    System.out.println("Please provide a single-digit number.");
  }
```

```
    System.out.println();
}
```

Code explanation

First, we create a new method named **printDiamondOutline** that accepts an integer named **size** as its parameter. We check to ensure that **size** is less than 10 before proceeding any further. Similar to the previous problem, double-digit numbers will complicate the required spacing, so we will avoid them for now. If we determine that **size** has a value of 10 or greater, we will print the string **Please provide a single-digit number**. and return to the calling program.

Inside the if statement previously, we will create a new for loop with a counter named **topCounter**. Again, we will initialize **topCounter** to a value of 1 as we only want to work with the positive integers less than or equal to size (5 in this example). We will increment **topCounter** and iterate through the loop as long as **topCounter** is less than or equal to size.

Next, we will compute and print the leading spaces for the top section of the diamond. We can do this with a simple for loop (created or nested inside of the loop mentioned above) that initializes a counter named **spaceCounter** to 0. We will increment **spaceCounter** and iterate through the loop (printing a single space) as long as **spaceCounter** is less than the value of size minus **topCounter**. This will generate a number of spaces that is inversely proportional to the value of **topCounter**.

We will then check to see if the value of **topCounter** is greater than 1. If it is not, then we have the top point of the diamond, so we can skip the (next) loop, print the value **topCounter** (without worrying about any filler spaces), and continue. If **topCounter** has a value that is indeed greater than 1, then we will print it and create a new for loop with a counter named **fillCounter** that is initialized to a value of 0. We will iterate through the loop incrementing **fillCounter**, as long as **fillCounter** has a value that is less than **topCounter** multiplied by 2, minus 3. This will generate the appropriate number of filler spaces before printing the value of **topCounter** as the outer edge of the diamond.

With the top half of the diamond printed, we will create a for loop

to bring the bottom half. It will have a counter named **bottomCounter** and initialized to a value of size minus 1. We will use size minus 1, because we do not want to print the middle row twice. We will decrement **bottomCounter** and iterate through the loop as long as **bottomCounter** is greater than 0.

We will then check to see if the value of **bottomCounter** is greater than 1. If it is not, then we have the bottom point of the diamond, so we can skip the (next) loop, and print the value of **bottomCounter** without any spacing. As we did in the top section, if **bottomCounter** has a value that is greater than 1, we will print it and create a new for loop with a counter named **fillCounter2** that is initialized to a value of 0. We will iterate through the loop incrementing **fillCounter2**, as long as **fillCounter2** has a value that is less than **bottomCounter** multiplied by 2, minus 3 (also as we did above).

When this is all complete, we should have our diamond outline as shown at the beginning of this problem. The important lesson with this (and the previous) program is to try to look for the mathematical patterns that help drive the output.

Program 1.6: Checking valid number using while

Write a program to accept a number as keyboard input and display the number's corresponding **American Standard Code for Information Interchange** (**ASCII**) or **Unicode Transformation Format** (**UTF**) character.

Algorithm

To run this program, we will take the following steps:

1. Create a new method named **readNumber**.
2. Instantiate variables for the following:
    a. The **Scanner** named **inputScanner**.
    b. The number read from the input named **number**, to the value of **9999**.
3. Create a while loop to run as long as **number != 0**, then:
    a. Prompt the user to enter a number greater than 31, or 0

to quit.

b. Read the number.

c. Convert the number from a string to an **Integer**.

d. If the number is greater than 31, print the ASCII/UTF character.

e. If the number is not greater than 31, but also is not 0, then indicate that the input is improper.

f. If the input is not a number, then indicate that the input is improper.

4. Close the **Scanner** object.

Flowchart

The flowchart for this problem can be seen in *Figure 1.7:*

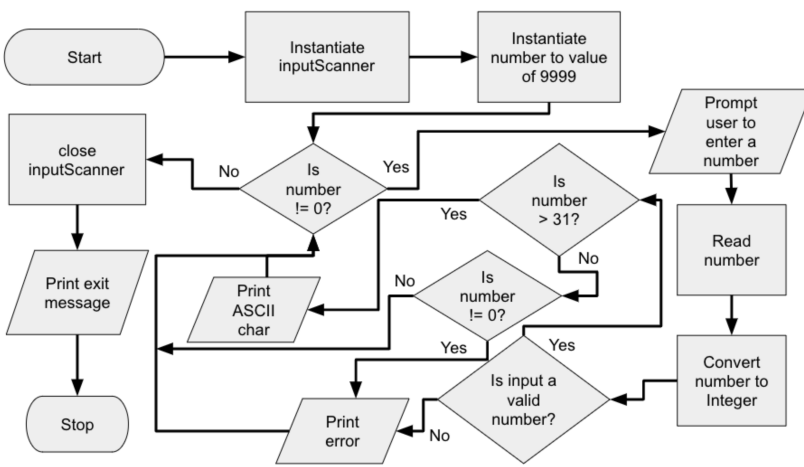

***Figure 1.7**: Flowchart showing the logic for the while valid number problem*

Java code

Here is the Java code for *Program 1.6:*

```
import java.util.Scanner;
private static void readNumber() {
    Scanner inputScanner = new Scanner(System.in);
    int number = 9999;
```

```
        while (number != 0) {
            System.out.println("Enter a number > 31,");
            System.out.print("or zero (0) to quit: ");
            String inputStr = inputScanner.nextLine();
            try {
                number = Integer.parseInt(inputStr);
                if (number > 31) {
                    System.out.printf("The character for ASCII code %d is
%c\n",
                            number, (char) number);
                } else if (number != 0) {
                    System.out.println(
                            "Sorry, only numbers 32 or higher are
permitted.");
                }
            } catch (NumberFormatException ex) {
                System.out.println("Sorry, only numbers are permitted.");
            }
        }
        inputScanner.close();
        System.out.println("0 entered, exiting.");
}
```

Code explanation

We begin by adding an import (at the top of our class definition) for the **java.util.Scanner** class, and then create a new method named **readNumber**. We will instantiate a **Scanner** object named **inputScanner** to read keyboard input. We will also instantiate an integer named **number** (before the loop) to a value of 9999.

Note: The number 9999 is not significant. Really, any non-zero integer will suffice.

Next, we will create a while loop. A for loop is a useful tool when

the number of iterations is known ahead of time. Since we do not really know how many times the user will want to iterate through the loop, a while loop is appropriate. We will define that loop with the condition to run as long as the **number** is not equal to 0. If a 0 is entered, we will use that as a signal to quit or exit the program.

Inside the loop, we will prompt the user to enter a number of 32 or greater, or 0 to quit. We will then read input from the keyboard and store the value in a **String** named **inputStr**. We will then implement a try/catch construct to handle any exceptions that arise from parsing **inputStr**.

Note: Numbers for UTF-16 codes run from 0 to 1,114,111. A good enhancement here would be to trap for a maximum number or ceiling. In our case, if the maximum value for an integer is exceeded, it will be caught by the NumberFormatException.

The first thing we do inside the try/catch is attempt to convert/parse **inputStr** to **number**. Assuming that an exception is not thrown, we then make sure that the number has a value greater than 31 and show an appropriate message if it is not. If the number is valid, then its ASCII/UTF code is shown in the output.

Note: ASCII/UTF codes as follows 32 are control codes that were mostly used to operate legacy printing hardware. As these codes do not always parse with consistent behavior, we are wisely restricting their use in this program.

At this point, the program flow returns to the beginning of the loop, and the condition for the **number** is checked. If the **number** is not equal to 0, the loop runs the logic described above for another iteration. If the **number** is equal to 0, the flow breaks out of the loop. The **inputScanner** reference is then closed, and an exit message is sent as output.

Program 1.7: Display user names

Write a program to accept usernames as keyboard input and display each user's name as they are entered.

Algorithm

To solve this program, we will take the following steps:

1. Create a new method named **readName**.
2. Instantiate variables for the following:
    a. The **Scanner** named **inputScanner**.
    b. A **String** to hold the result of **inputScanner.nextLine()** named **inputStr**.
3. Prompt the user to enter their name.
4. Read in their name.
5. Create a do-while loop containing the following steps:
    a. Print a message greeting the user with their name.
    b. Prompt the user to enter their name.
    c. Read in their name.
    d. Check that **inputStr** does not equal **exit**.
6. Close the **Scanner** object.

Flowchart

The flowchart for this problem can be seen in *Figure 1.8*:

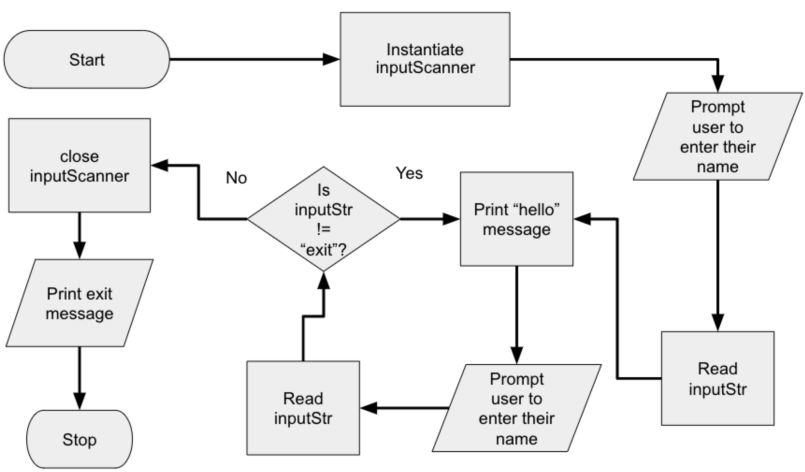

***Figure 1.8**: Flowchart showing the logic for the display usernames problem*

Java code

Here is the Java code for *Program 1.7*:

```
private static void readName() {
    Scanner inputScanner = new Scanner(System.in);
    System.out.println("What is your name?");
    String inputStr = inputScanner.nextLine();
    do {
        System.out.printf("Hello %s!\n", inputStr);
        System.out.println("What is your name?");
        inputStr = inputScanner.nextLine();
    } while (!inputStr.equals("exit"));
    inputScanner.close();
    System.out.println("Exiting.");
}
```

Code explanation

Similar to the last problem, we will want to make sure that the **java.util.Scanner** class has been imported. We will also want to create a new method named **readName**. We will need a variable instantiated for the **Scanner** named **inputScanner**. We will start by prompting the user for their name and use the scanner to read their input into a **String** named **inputStr**.

We will then use a do-while loop. The difference between a do-while loop and a while loop is that the do-while loop checks the condition at the end, ensuring that the loop will at least run once. Inside the loop, we will print a greeting to the user, assuming their name is the value of **inputStr**. We will then prompt them for their name (again) and re-read it into **inputStr**.

At this point, we will check the value of **inputStr**, and the loop will re-run as long as **inputStr** does not have a **String** value of exit. Once a value of exit is read, the loop will not re-run; we will close **inputScanner** and print a closing message.

Conclusion

In this chapter, we introduced three kinds of loops, as well as examples for their use. When choosing a loop construct, it is

important to understand if the number of iterations to be performed is known (or can be computed) ahead of time. It is also important to consider if the code inside the loop runs at least once or not at all.

We also introduced the concept of flowcharting as a way to visually describe progressions of programming logic. As this book's lessons advance in complexity, understanding the flowchart process will be important.

In the next chapter, we will present different ways to work with files from Java code. These problems will exercise fundamental structures and techniques such as reading and writing from files, working with common types of text files, and utilizing libraries for masking sensitive data.

Points to remember

Have a look at some key points to remember:

- For loops are useful when the number of loop iterations is known ahead of time.
- For-each loops are much easier to use and understand when working with collections.
- While loops are useful when the number of loop iterations is unknown, or if there are multiple conditions that the loop is dependent on.
- Do-while loops are useful with an unknown number of loop iterations, and when the loop code should run at least once.
- In Java, dependent libraries are imported at the top of the class definition.
- The print and println functions differ in that only the latter automatically includes a carriage return. They can be used together to build specific output statements.

Questions

Answer the following questions to check your understanding:

1. Could Program 1.1 (Print the contents of an array) be

rewritten without a for-each loop? What would be the advantages or disadvantages of doing so?

2. Could Program 1.2 (Reverse an array) be rewritten with a for-each loop? Why or why not?
3. The code presented for Program 1.3 (FizzBuzz) shows the problem being solved with three if constructs. Could it be solved with fewer than that? Why or why not?
4. Currently, Programs 1.4 (Print a diamond of numbers) and 1.5 (Diamond outline of numbers) both limit the size of the diamond to a single-digit number. What would have to change for them to work with double-digit numbers?
5. Program 1.6 (While valid number) currently uses a default value for number of 9999. Can you think of a different default condition for the while loop to properly function?
6. Program 1.7 (Display user names) is currently written with a do-while loop. Can you think of a way to solve it with a standard while loop?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 2

File Operations

Introduction

In this chapter, we will tackle some simple problems using files. Many tasks that enterprise Java developers encounter involve writing and reading data to and from files. Thus, the problems within will aim to adhere to those that may be encountered in the real-world throughout the book.

Structure

In this chapter, we will cover the following topics:

- Program 2.1: Write to a text file
- Program 2.2: Read and print from a text file
- Program 2.3: Read and print from a CSV file
- Program 2.4: Write a password file
- Program 2.5: Validate a user from a password file
- Program 2.6: Write a unique password file

Objectives

The programming problems in this chapter will present different ways to work with files from Java code. These problems will exercise fundamental structures and techniques such as reading and writing from files, working with common types of text files, and utilizing libraries for masking sensitive data. By the end of this chapter, we should have a good understanding of how to work with different kinds of text files and data.

Program 2.1: Write to a text file

Write a program to read a list of strings and write each item into a text file. We can test it with this code:

```
String dataFile1 = "data1.txt";
List<String> atariGames = Stream.of(
        "Pac-Man", "Space Invaders", "Donkey Kong",
        "Pitfall!", "Frogger"
        ).toList();
writeListToFile(dataFile1, atariGames);
```

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate variables for the list of strings and the name of the file. Then pass them both as parameters to the new method named **writeListToFile**.
2. Define a local variable named **writer** of type **BufferedWriter**. Create a new instance of it by passing a new **FileWriter** object with a file location/name as its lone parameter.
3. Loop through the list of strings. Write each item into the file.
4. Close the **writer**.
5. Print a completion message.
6. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 2.1*:

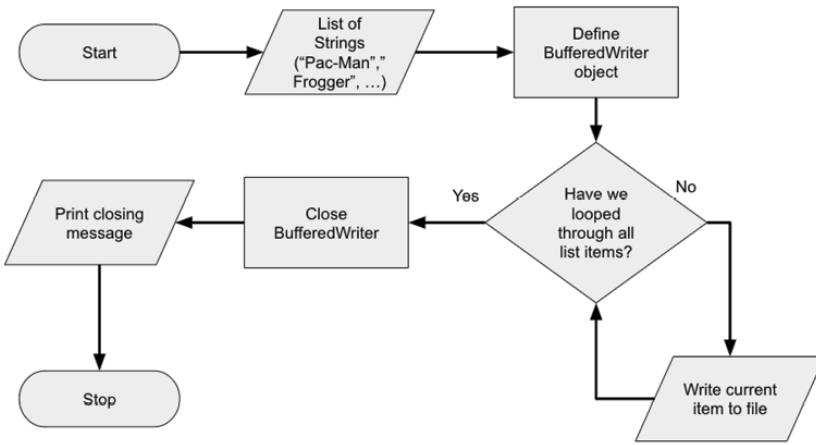

***Figure 2.1**: Flowchart describing how to write list items to a simple text file*

Java code

The Java code for *Program 2.1* is specified as follows:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;
private static void writeListToFile(String filename, List<String>
list) {
    try {
        BufferedWriter writer =
                new BufferedWriter(new FileWriter("data/" + filename));
        for (String item: list) {
            writer.write(item + "\n");
        }
        writer.close();
    } catch (IOException ioex) {
```

```
            System.out.println("Error occurred while writing:");
            ioex.printStackTrace();
        }
        System.out.println(filename + " written!");
    }
```

Code explanation

For this program, we will need to import several libraries. The other programs will end up using many of the same libraries, but for this program specifically, we will need to import the following:

- **From** java.io:

java.io.BufferedWriter
java.io.FileWriter
java.io.IOException

- **From** java.util:

java.util.List

We will start out by creating a new method named **writeListToFile**. It will accept the name of a file (**filename**) and a list of strings (**list**) as parameters. Inside our method, we will need to wrap our file operations in a try/catch construct, as the **BufferedWriter** and **FileWriter** constructors and object methods will throw an **IOException**.

We will define a new **BufferedWriter** with a nested type of **FileWriter**, and name it **writer**. The **writer** will write to the file specified by **filename** in the local data directory. A **FileWriter** by itself would probably work just fine for our small example, but almost all real-world problems involving writing to files will benefit from using the **BufferedWriter**. The **BufferedWriter** utilizes an internal buffer that helps to abstract the disk writing process, and ultimately results in fewer disk operations, thus increasing performance.

We will use a for each loop to iterate through the items in **list**. Inside the loop, we will call the write method from the **writer** object to persist each **item** to our file. Once the for loop completes, we will call the **close** method from **writer** to gracefully release the file handle and its resources.

Assuming that an **IOException** has not been thrown, we will print a simple message to indicate that our file has been successfully written. Our method will then return to the calling method.

Program 2.2: Read and print from a text file

Write a program to read a small text file and print each line. We can test it with the file from *Program 2.1*, like this:

```
readTextFile(dataFile1);
```

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate a variable for the name of the file. Then pass it as a parameter to the new method named **readTextFile**.
2. Define a local variable named **reader** of type **BufferedReader**. Create a new instance of it by passing a new **FileReader** object with a file location/name as its lone parameter.
3. Define another local variable named **line** of type **String** and instantiate it with the result of calling the **reader.readLine()** method.
4. While **line** is not equal to null, print the value of **line** and then read the next line by calling **reader.readLine()** again.
5. Once **line** is equal to null, break out of the loop and close **reader**.

Flowchart

The flowchart for this problem can be seen in *Figure 2.2*:

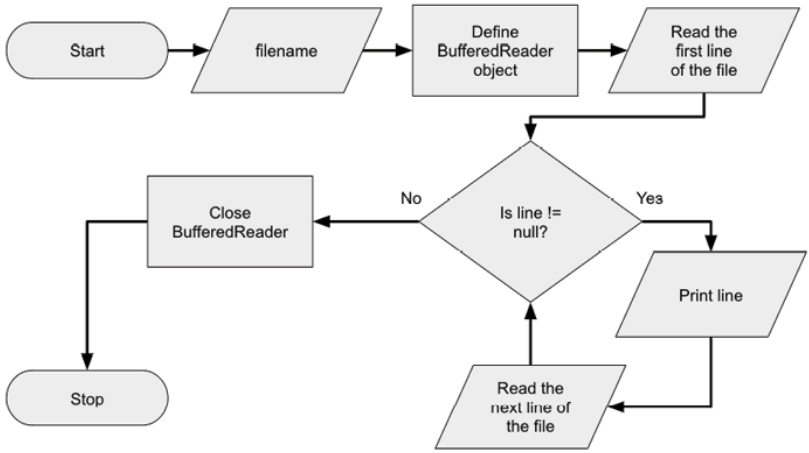

***Figure 2.2**: Flowchart describing how to read the content of a text file*

Java code

The Java code for *Program 2.2 is* specified as follows:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
private static void readTextFile(String filename) {
   try {
      BufferedReader reader =
            new BufferedReader(new FileReader("data/" + filename));
      String line = reader.readLine();
      while (line != null) {
         System.out.println(line);
         line = reader.readLine();
      }
      reader.close();
   } catch (IOException ioex) {
      System.out.println("Error occurred while reading:");
      ioex.printStackTrace();
```

```
    }
}
```

Code explanation

For this program, we will need to import a few libraries. Specifically, we will need to import:

- **From** java.io**:**

java.io.BufferedReader
java.io.FileReader
java.io.IOException

We will start out by creating a new method named **readTextFile**. It will accept the name of a file (**filename**) as its only parameter. Inside our method, we will need to wrap our file operations in a try/catch construct, as the **BufferedReader** and **FileReader** constructors and object methods will throw an **IOException**.

We will define a new **BufferedReader** with a nested type of **FileReader**, and name it **reader**. The **reader** will read from the file specified by **filename** in the local data directory. As mentioned in the prior program, **FileReader** by itself would probably work just fine for our small example, but almost all real-world problems involving writing to files will benefit from using the **BufferedReader**.

We will define a local **String** named **line**, and instantiate it with the result of the **reader.readLine()** method. We will then use a while loop to print the value of **line** and then read the next line in the file with another call to **reader.readLine()**.

Once **line** reads the value of null, the loop will exit. Assuming that an **IOException** has not been thrown, we will return to the calling method.

Program 2.3: Read and print from a CSV file

Write a program to read a small **comma-separated value** (**CSV**) file and print each line. Specifically, our method will read a CSV file of video arcade games, players, and their scores, and print them to the screen. We can test it with the following code:

```
String csvFile1 = "arcadeScores.csv";
List<String> arcadeScores = Stream.of(
        "Game,Player,Score",
        "Pac-Man,Aaron,188870", "Donkey Kong Jr.,Emily,63000",
        "BurgerTime,Aaron,313650", "Galaga,Avery,34520",
        "Super Mario Brothers,Coriene,239400"
        ).toList();
writeListToFile(csvFile1, arcadeScores);
System.out.println();
readCSVFile(csvFile1);
```

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate a variable for the name of the file. Then pass it as a parameter to the new method named **readCSVFile**.
2. Define a local variable named **reader** of type **BufferedReader**. Create a new instance of it by passing a new **FileReader** object with a file location/name as its lone parameter.
3. Define a local variable named **line** of type **String** and instantiate it with the result of calling the **reader.readLine()** method.
4. Define a local Boolean variable named **header**, and instantiate it with a value of **true**.
5. Execute the loop while **line** is not equal to null.
6. Inside the loop, check if **header** is not equal to true. If so, split the line into columns delimited by a comma(‘,’). Assign each column value to a local **String** variable (**game**, **player**, **score**) and print them all on a single line with some additional, supporting text.
7. If **header** is true, then simply print the value of **line**.
8. Once **line** is equal to null, break out of the loop and close **reader**.

Flowchart

The flowchart for this problem can be seen in *Figure 2.3*:

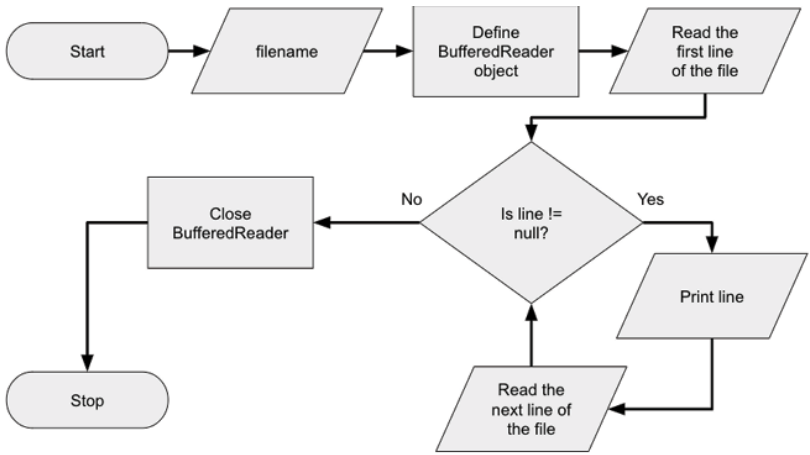

***Figure 2.3**: Flowchart describing how to read the content of a CSV file*

Java code

The Java code for *Program 2.3* is specified as follows:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
private static void readCSVFile(String filename) {
    try {
        BufferedReader reader =
                new BufferedReader(new FileReader("data/" + filename));
        String line = reader.readLine();
        // the first line in the file should be the header row
        boolean header = true;
        while (line != null) {
            if (!header) {
                String[] columns = line.split("[,]");
                String game = columns[0];
```

```
            String player = columns[1];
            String score = columns[2];
            System.out.printf("High score for %s ", game);
            System.out.printf("by player %s of ", player);
            System.out.printf("%s points\n", score);
         } else {
            // print header row
            System.out.println(line);
            header = false;
         }
         line = reader.readLine();
      }
      reader.close();
   } catch (IOException ioex) {
      System.out.println("Error occurred while reading:");
      ioex.printStackTrace();
   }
}
```

Code explanation

For this program, we will need to import a few libraries. Specifically, we will need to import:

- **From** java.io**:**

java.io.BufferedReader
java.io.FileReader
java.io.IOException

We will start out by creating a new method named **readCSVFile**. It will accept the name of a file (**filename**) as its only parameter. Inside our method, we will need to wrap our file operations in a try/catch construct, as the **BufferedReader** and **FileReader** constructors and object methods will throw an **IOException**.

We will define a new **BufferedReader** with a nested type of

FileReader, and name it **reader**. The **reader** will read from the file specified by **filename** in the local data directory. As mentioned in the prior programs, **FileReader** by itself would probably work just fine for our small example, but almost all real-world problems involving writing to files will benefit from using the **BufferedReader**.

We will define a local **String** named **line**, and instantiate it with the result of the **reader.readLine()** method. Additionally, we will define a local Boolean value named **header**, and instantiate it with a value of **true**. This is because the first line in our file will be a header row, and we will want to process that line differently.

We will then use a while loop to iterate through the file until we reach the end (no more readable lines). Inside the loop, we will first check if the **header** does not have a value of true. If statements perform better when the condition that occurs most often is coded first. As we will only process the header row once, we will handle that in our else condition.

If we are indeed not working with a header row, we will **split()** the line into a **String** array named **columns**, and assign each column to a local variable (game, player, score). We will then format and print each game's player name and high score value on a single line. If we are working with a header row, we will simply print the value of **line** and set **header** to a value of **false**. Once we have made it through the if statement, we will read the next line in the file with another call to **reader.readLine()**.

Once **line** reads the value of null, the loop will exit. Assuming that an **IOException** has not been thrown, we will close **reader** and return to the calling method.

The split() method takes a regular expression as a parameter. The regular expression pattern that we are using is simply: [,]

Program 2.4: Write a password file

Write a program to accept a username and password combination, encrypt the password, and write them to a file.

We will test our program with the following code:

```
writeToPasswordFile("aaron", "Brewers82");
writeToPasswordFile("coriene", "loveBug22");
writeToPasswordFile("avery", "rumbleBoy58");
writeToPasswordFile("christian", "BHappy42");
writeToPasswordFile("emily", "chimPoKoMon58");
writeToPasswordFile("victoria", "ethanIsAwesome77");
```

After the file has been written to multiple times, its contents should look something like this:

```
aaron,$2a$10$8FXsQ0YbH2j/
Ha75nPFDi.gRldGgAXLTEEaMByAoCPDdOFbTR/ftG
coriene,$2a
$10$EbzpFkcbLerGP.FMyWfJEufTd/7FZJ04gw5KkNtn06oHpZ7ZzMsOi
avery,$2a
$10$XNkD7PKO98noZeDiecvjYOFRX2.FY0gssjXFgsZgsXEnQYHVgOrKi
christian,$2a$10$GawTLqJSxMstgLzQQvaz3eY2Hg6oHG/
oXPf.rnmlfHOguGyy03/fW
emily,$2a
$10$ExzXZLAVhdds2vIsPc5LHeC3TbdaZbvrwzKVdLNPWxp1HqLpotwKe
victoria,$2a
$10$0O5tQnxCLsxUUB2dWDGibO8FbINWXCQRzrXOMoJ89cqP.asINDYU
```

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate variables for a username and password and then pass them both as parameters to a new method named **writeToPasswordFile**.
2. Define a local variable named **writer** of type **BufferedWriter**. Create a new instance of it by passing a new **FileWriter** object with a file location/name as the first parameter. The second parameter should be the Boolean literal of **true**, indicating that the file should be opened in **append** mode.
3. Define a local variable named **pEncoder** of type **BCryptPasswordEncoder**.

4. Use the **BCryptPasswordEncoder**'s **encode()** method to create an encoded version of the password, and store it in a local **String** variable named **hashedPassword**.
5. Write **username** and **hashedPassword** to the file.
6. Close the **writer**.
7. Print a completion message.
8. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 2.4*:

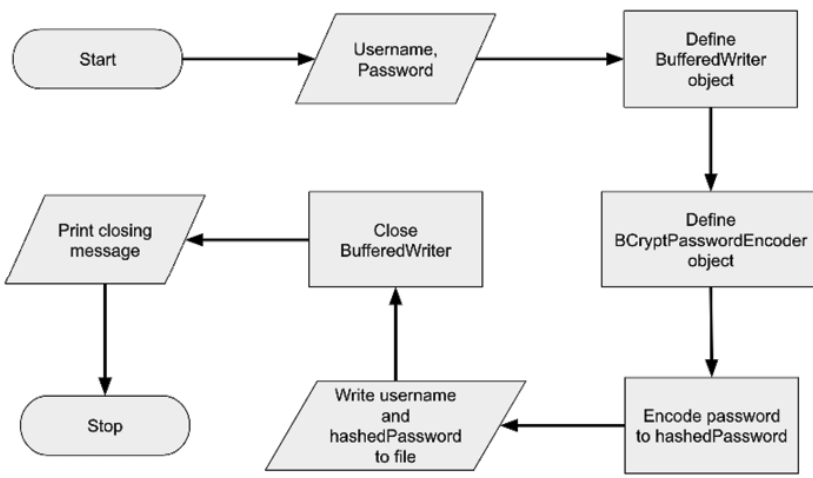

***Figure 2.4**: Flowchart describing how to write a password file*

Java code

The Java code for *Program 2.4* can be seen as follows:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
private static void writeToPasswordFile(String username, String
password) {
```

```
    try {
        // open file with append == true
        BufferedWriter writer =
                new BufferedWriter(new FileWriter("data/
password.csv",true));
        // Hash the password
        BCryptPasswordEncoder pEncoder = new
BCryptPasswordEncoder();
        String hashedPassword = pEncoder.encode(password);
        writer.write(username + "," + hashedPassword + "\n");
        writer.close();
    } catch (IOException ioex) {
        System.out.println("Error occurred while writing:");
        ioex.printStackTrace();
    }
    System.out.println(username + " written to password file.");
}
```

Code explanation

Similar to the prior programs in this chapter, this program will require imports for the appropriate libraries for writing to files and handling a possible input/output exception. We will also need to import a specific Security library from the Spring Framework. This library will help us to properly encode our users' passwords prior to storage on the filesystem.

Note: Sensitive information like user passwords or other credentials should never be stored (in a file or in a database) as plain text.

The following libraries should be imported:

- **From** java.io:

java.io.BufferedWriter
java.io.FileWriter
java.io.IOException

- **From** org.springframework.security:

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder

Prior to using the **BCryptPasswordEncoder**, the Spring Security library will need to be specified in the project's dependency management software (ex, Maven, Gradle). The following properties will allow the dependencies for this library to be properly configured:

- **Group ID**: **org.springframework.security**
- **Artifact ID**: **spring-security-web**
- **Version**: **6.4.2**

We will start out by creating a new method named **writeToPasswordFile**. It will accept a username and a password (both of type **String**) as its parameters. Inside our method, we will need to wrap our file operations in a try/catch construct, as the **BufferedWriter** and **FileWriter** constructors and object methods will throw an **IOException**.

We will define a new **BufferedWriter** with a nested type of **FileWriter**, and name it **writer**. The **writer** will write to the **data/password.csv** file specified as a parameter in the call to the (nested) **FileWriter** constructor. The file should also be opened in **append** mode, which requires the Boolean literal of **true** to be passed as an additional parameter.

Note: Specifying true for append mode will allow each subsequent call to this method to write a new line to the end of the file.

Next, we will define a local variable named **pEncoder** of type **BCryptPasswordEncoder**. This will allow us to access the encoding functionality. We will then use its **encode()** method to hash the password using the (Spring 2024) bcrypt algorithm. This encoded version of the password will be stored locally in a String variable named **hashedPassword**.

The **username** and **hashedPassword** will then be written to a CSV file. Finally, we will call the **close()** method on **writer** and return to the calling program.

Note: The bcrypt password hashing process is deliberately slow by design. This provides an additional obstacle for brute force attacks on password security.

Program 2.5: Validate a user from a password file

Write a program to authenticate a user with their password.

We will test our program with the following code:

```
validateUserFromFile("aaron","Brewers82");
validateUserFromFile("avery","blahblahblah");
```

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate variables for a username and a password and then pass them both as parameters to a new method named **validateUserFromFile**.
2. Define a local variable named **reader** of type **BufferedReader**. Create a new instance of it by passing a new **FileReader** object with a file location/name as the lone parameter.
3. Define a local variable named **pEncoder** of type **BCryptPasswordEncoder**.
4. Define a local Boolean variable named **found**, and instantiate it with a value of **false**. This variable will help us determine if the value of **username** was found in the file.
5. Read the first line of the file and store it in a local **String** variable named **line**.
6. Create a while loop to run so long as **line** is not equal to null.
7. Inside the loop, begin by splitting **line** into an array of **String** named **columns**.
8. Instantiate a local **String** variable named **fileUser** with the value of **columns[0]**. This variable name is prefaced with the word **file**, to indicate that it has been read from the file, so as not to be confused with **username**.
9. Create an if/else construct, checking whether or not the **String** values of **username** and **fileUser** are equal.

10. If the **String** values of **username** and **fileUser** are equal, set **found** to a value of **true**. Create another if/else construct to check whether or not the hashed password value stored in **columns[1]** is equal to **password**. This can be done with the **pEncoder.matches()** method. Print a message indicating whether or not the passwords matched, and then **break** out of the loop.
11. If the **String** values of **username** and **fileUser** are not equal, then read the next line from the file. If the result of **reader.readLine()** is null, the loop will exit.
12. Close **reader**.
13. If **username** could not be found in the file, indicate this with a printed message.
14. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 2.5*:

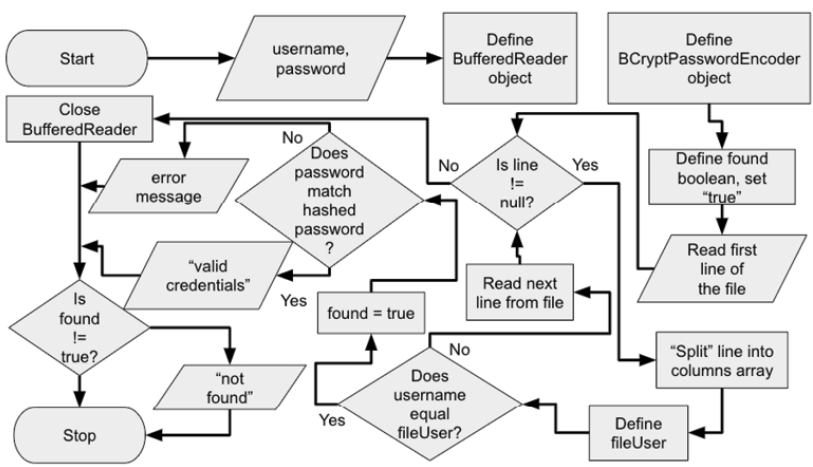

***Figure 2.5**: Flowchart describing how to validate a user with a password file*

Java code

The Java code for *Program 2.5* can be seen as follows:

```
import java.io.BufferedReader;
```

```
import java.io.FileReader;
import java.io.IOException;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
private static void validateUserFromFile(String username, String
password) {
   try {
      BufferedReader reader =
            new BufferedReader(new FileReader("data/
password.csv"));
      BCryptPasswordEncoder pEncoder = new
BCryptPasswordEncoder();
      boolean found = false;
      String line = reader.readLine();
      while (line != null) {
         String[] columns = line.split("[,]");
         String fileUser = columns[0];
         if (fileUser.equals(username)) {
            found = true;
            String fileHashedPwd = columns[1];
            if (pEncoder.matches(password, fileHashedPwd)) {
               System.out.printf("User %s has valid credentials.\n",
                     username);
            } else {
               System.out.printf("User %s could not be authenticated.
\n",
                     username);
            }
            break;
         }
         line = reader.readLine();
```

```
        }
        if (!found) {
            System.out.printf("User %s was not found.\n", username);
        }
        reader.close();
    } catch (IOException ioex) {
        System.out.println("Error occurred while reading:");
        ioex.printStackTrace();
    }
}
```

Code explanation

As with our prior programs, this program will require imports for the appropriate libraries for writing to files and handling a possible input/output exception. We will also need to import the Security library from the Spring Framework.

Note: Sensitive information like user passwords or other credentials should never be stored (in a file or in a database) as plain text.

The following libraries should be imported:

- **From** java.io:

java.io.BufferedReader
java.io.FileReader
java.io.IOException

- **From** org.springframework.security:

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder

Likewise, prior to using the **BCryptPasswordEncoder**, the Spring Security library will need to be specified in the project's dependency management software. The following properties will allow the dependencies for this library to be properly configured:

- **Group ID**: **org.springframework.security**
- **Artifact ID**: **spring-security-web**

Version: 6.4.2

We will start out by creating a new method named **validateUserFromFile**. It will accept a username and a password (both of type **String**) as its parameters. Inside our method, we will need to wrap our file operations in a try/catch construct, as the **BufferedReader** and **FileReader** constructors and object methods will throw an **IOException**.

We will define a new **BufferedReader** with a nested type of **FileReader**, and name it **reader**. The **reader** will read from the **data/password.csv** file specified as a parameter in the call to the (nested) **FileReader** constructor.

Next, we will define a local variable named **pEncoder** of type **BCryptPasswordEncoder**. This will allow us to access the matching functionality. We will also instantiate a local, Boolean variable named **found** with the value of **false**. The last local variable required at this point is for a **String** value named **line**, which we will instantiate by reading the first line in the file by calling **reader.readLine()**.

We will then create a while loop, defined to run so long as the value of **line** is not equal to null. Inside the loop, we will **split** the **String** into an array of **String** named **columns**. The **columns** array will hold data for only one line at a time and should only ever have two columns: one for the username and the other for the hashed password.

We will then instantiate a local **String** variable named **fileUser** with the value of **columns[0]**. This should keep us from making the mistake of using the wrong indexed value from **columns**. Additionally, it will serve the purpose of clearly defining the user we are checking for (**username**) and the user that we are checking **against** (**fileUser**).

If **fileUser** and **username** contain **String** values that are equal, we will set **found** to **true** and instantiate a new **String** variable named **fileHashedPwd** to the value of **columns[1]**. We will then call **pEncoder.matches()** to see if the **String password** matches the hashed password stored in the **String fileHashedPwd**. Depending on whether or not the result of **pEncoder.matches()** is true, we will print an appropriate message. We will then use Java's **break**

command to exit from the while loop early. At this point, we know that we have found our user, so there is no need to read each remaining line in the file.

Note: The same password hashed twice will produce two different results. This is because bcrypt includes a random value (also known as a salt) with the password string, to help prevent the hashed value from being reverse-engineered. Therefore, the matches method is the only way of telling whether or not a password and a hashed password have the same underlying String value.

If **fileUser** and **username** contain **String** values that are not equal, we will read the next line in the file into the **String** variable **line**. The logic flow of the program will then return to the beginning of the loop.

Once we are done with the while loop, we will check if the value of **found** is false and print an appropriate message. Finally, we will close **reader** and return to the calling program.

Program 2.6: Write a unique password file

Write a program to accept a username and password combination, encrypt the password, and write them to a file, as long as the username does not already exist inside the file.

We will test our program with the following code:

```
writeToPasswordFileUnique("aaron", "Brewers82");
writeToPasswordFileUnique("coriene", "loveBug22");
writeToPasswordFileUnique("avery", "rumbleBoy58");
writeToPasswordFileUnique("coriene", "loveBug22");
writeToPasswordFileUnique("emily", "chimPoKoMon58");
writeToPasswordFileUnique("victoria", "ethanIsAwesome77");
```

After the file has been written to multiple times, its contents should look something like this:

```
aaron,$2a$10$8FXsQ0YbH2j/
Ha75nPFDi.gRldGgAXLTEEaMByAoCPDdOFbTR/ftG
coriene,$2a
$10$EbzpFkcbLerGP.FMvWfJEufTd/7EZJ04gw5KkNtn06oHpZ7ZzMsOi
```

```
avery,$2a
$10$XNkD7PKO98noZeDiecvjYOFRX2.FY0gssjXFgsZgsXEnQYHVgOrKi
emily,$2a
$10$ExzXZLAVhdds2vIsPc5LHeC3TbdaZbvrwzKVdLNPWxp1HqLpotwKe
victoria,$2a
$10$0O5tQnxCLsxUUB2dWDGibO8FbINWXCQRzrXOMoJ89cqP.asINDYU
```

The important takeaway from the previous is that the user *Coriene* should not be listed in the file twice.

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate variables for a username and a password and then pass them both as parameters to a new method named **writeToPasswordFileUnique**.
2. Define a local variable named **currentUsers** as a new, empty **ArrayList**.
3. Implement a try/catch construct with two catch clauses. The first will be for the **FileNotFoundException**, and the second for the **IOException**. If a **FileNotFoundException** is encountered, we should print a message indicating this, but allow the program to continue.
4. Inside the try construct, define a local variable named **reader** of type **BufferedReader**. Create a new instance of it by passing a new **FileReader** object with a file location/name as the lone parameter.
5. Define a local **String** variable named **line**, and instantiate it with the result of **reader.readLine()**.
6. Create a while loop to iterate as long as **line** does not have a value of null. This loop will split **line** into a String array named **columns**. It will then add the value of **columns[0]** (username) to the **currentUsers** list. Once we have read through the file and added all users to **currentUsers**, the loop will end, and we can continue.
7. Inside another try/catch, we will check to see whether or not the **currentUsers** list **contains()** the value of **username**. If it

does, then we do not need to add it again, and we will return to the calling program.

8. If the user does not exist, we will define a local variable named **writer** of type **BufferedWriter**. Create a new instance of it by passing a new **FileWriter** object with a file location/name as the first parameter. The second parameter should be the Boolean literal of **true**, indicating that the file should be opened in **append** mode.
9. Define a local variable named **pEncoder** of type **BCryptPasswordEncoder**.
10. Use the **BCryptPasswordEncoder**'s **encode()** method to create an encoded version of the password and store it in a local **String** variable named **hashedPassword**.
11. Write **username** and **hashedPassword** to the file.
12. Close the **writer**.
13. Print a completion message.
14. Return to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 2.6*:

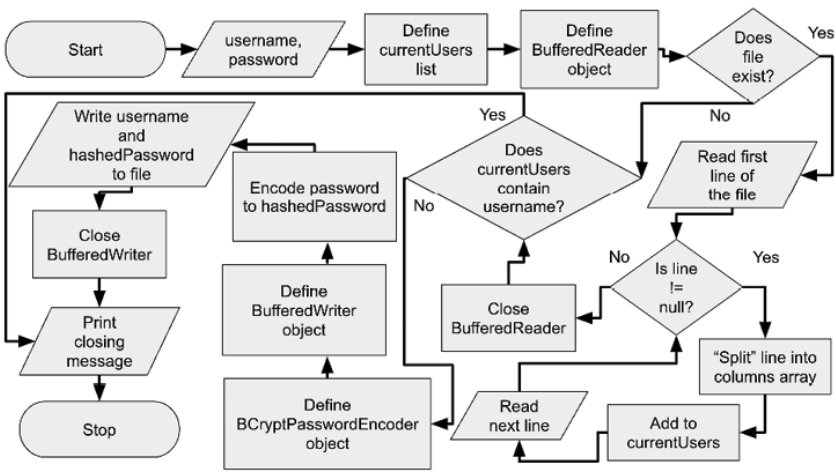

***Figure 2.6**: Flowchart describing how to write unique users to a password file*

Java code

The Java code for *Program 2.6* can be seen as follows:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader
import java.io.FileWriter;
import java.io.FileNotFoundException;
import java.io.IOException;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import java.util.ArrayList;
import java.util.List;
private static void writeToPasswordFileUnique(String username,
String password) {
    List<String> currentUsers = new ArrayList<>();
    // open file reader
    try (BufferedReader reader =
            new BufferedReader(new FileReader("data/passwordU.csv")))
{
        String line = reader.readLine();
        while (line != null) {
            String[] columns = line.split("[,]");
            currentUsers.add(columns[0]);
            line = reader.readLine();
        }
        reader.close();
    } catch (FileNotFoundException fnfex) {
        System.out.println("File was not found; must be first user.
Adding...");
    } catch (IOException ioex) {
        System.out.println("Error occurred while writing:");
```

```
        ioex.printStackTrace();
    }
    // open file with append == true
    try (BufferedWriter writer =
            new BufferedWriter(new FileWriter("data/
passwordU.csv",true))) {
        if (currentUsers.contains(username)) {
            System.out.printf("User %s exists, credentials not written.\n",
                    username);
        } else {
            // Hash the password
            BCryptPasswordEncoder pEncoder = new
BCryptPasswordEncoder();
            String hashedPassword = pEncoder.encode(password);
            writer.write(username + "," + hashedPassword + "\n");
            writer.close();
            System.out.println(username + " written to password file.");
        }
    } catch (IOException ioex) {
        System.out.println("Error occurred while writing:");
        ioex.printStackTrace();
    }
}
```

Code explanation

Similar to the prior programs in this chapter, this program will require imports for the appropriate libraries for reading and writing to files, as well as handling two possible file exceptions. We will also need to import the Security library from the Spring Framework, as well as two libraries to work with list-based collections.

The following libraries should be imported:

- **From** java.io:

java.io.BufferedReader
java.io.BufferedWriter
java.io.FileReader
java.io.FileWriter
java.io.FileNotFoundException
java.io.IOException

- **From** org.springframework.security:

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder

- **From** ava.util:

java.util.ArrayList
java.util.List

Before using the **BCryptPasswordEncoder**, the Spring Security library will need to be specified in the project's dependency management software (ex, Maven, Gradle). The following properties will allow the dependencies for this library to be properly configured:

- **Group ID**: **org.springframework.security**
- **Artifact ID**: **spring-security-web**
- **Version**: **6.4.2**

We will start out by creating a new method named **writeToPasswordFileUnique**. It will accept a username and a password (both of type **String**) as its parameters. Inside our method, we will start by creating a new, empty **ArrayList** (of type **String**) named **currentUsers**. As before, we will create a try/catch construct to wrap our read operation, as the **BufferedReader** and **FileReader** constructors and object methods will throw an **IOException**. We will name the password file **data/passwordU.csv**.

Note: By enclosing our buffered reader and buffered writer definitions in parentheses with the try construct, we are essentially using a try-with-resources approach. This way, if any of the file operations throw an exception, the reader and writer will be automatically closed. Although we will leave our explicit close() method calls in the code, just as a reminder.

When we are adding our first user to the password file, it will not yet exist. This will cause a **FileNotFoundException** to be thrown. As **FileNotFoundException** inherits from **IOException**, its catch block

will need to precede our catch block for **IOException**. The difference for this program's logic is that we still want the program to continue in the event of a **FileNotFoundException**.

Note: A FileNotFoundException is a type of IOException.

We will define a local **String** variable named **line** and instantiate it with the first line read from the file. That will be followed by a short while loop, which will run as long as **line** is not equal to null. Essentially, this loop will read the comma-separated line from the file, split out the user (**column[0]**), and add that user to the **currentUsers** list. Once the loop is complete, we will close the file reader and continue.

Next, we will check **currentUsers** to see if the String value of **username** is already in the file. We can do this using the **contains()** method. If the user already exists, we will print a message indicating this and return to the calling program.

If **username** does not exist in **currentUsers**, we will define a new **BufferedWriter** with a nested type of **FileWriter**, and name it **writer**. The **writer** will write to the **data/passwordU.csv** file specified as a parameter in the call to the (nested) **FileWriter** constructor. The file should also be opened in **append** mode, which requires the Boolean literal of **true** to be passed as an additional parameter.

Next, we will define a local variable named **pEncoder** of type **BCryptPasswordEncoder**. We will then use its **encode()** method to hash the password. This encoded version of **password** will be stored locally in a **String** variable named **hashedPassword**.

The **username** and **hashedPassword** will then be written to a CSV file. Finally, we will call the **close()** method on **writer** and return to the calling program.

Conclusion

In this chapter, we discussed reading from and writing to files. We introduced a few Java libraries that help us work with files more efficiently, including how to trap for certain types of file-based exceptions. We also worked with different types of files, including text and CSV, and learned how to process them. Finally, we covered

a few use cases for sensitive credential data and showed how to hash and store passwords properly.

The lessons learned in this chapter may be on a smaller scale. However, they should provide a solid foundation of learning for some of the more challenging problems ahead, including those faced by professional software developers.

In the next chapter, the readers will gain a working understanding of Character and String data types. This includes challenges with reading, storing, and changing text data, as well as how to leverage many of the Character and String class methods to achieve desired results.

Points to remember

Have a look at some key points to remember:

- Any file stream object that is opened must also be closed.
- Objects instantiated from the BufferedReader/Writer and FileReader/Writer classes require exception handling.
- The BufferedReader/Writer classes utilize an internal buffer that helps to abstract disk access, and ultimately results in fewer disk operations, thus increasing performance.
- Lines read from a CSV file can be separated into different columns using the split() method from the String class.
- FileWriters opened for writing can be appended to by passing a Boolean value of true for the append parameter.
- Sensitive data like access credentials and passwords should never be stored (in a file or in a database) as plain text.
- The bcrypt hashing algorithm uses an automatic salt (a String of random values) to help prevent hashed passwords from being reverse-engineered.
- The bcrypt algorithm is intentionally slow to work as an obstacle against brute force password attacks.

Questions

Answer the following questions to check your understanding:

1. What is the benefit of the BufferedReader/Writer over the FileReader/Writer?
2. What does the readLine() method (from BufferedReader and FileReader) return at the end of a file?
3. The split() method used in this chapter works by using the pattern [,] which is a regular expression. As this regular expression only matches on a single column, what kinds of problems might we encounter with our file data?
4. If we ran the encode() method (from the BCryptPasswordEncoder) library on the same String value twice, would the output be the same?
5. Which Java class does the FileNotFoundException inherit from?
6. Program 2.6 (Write a unique Password File) reads the entire passwordU.csv file every time that it is called, which is not very efficient. What are some more efficient ways that we could use to populate the currentUsers list?

CHAPTER 3

Characters and Strings

Introduction

This chapter's focus will be on the methods and techniques used to work with Character and String data types. Processing Character and String data is a common task that many software developers execute daily, and understanding how to work with them can be the difference between writing poor and high-performing code.

In Java, there are two kinds of variables: primitive types and reference types. With primitive types, their data is stored in memory as-is. Reference types, on the other hand, are essentially bookmarks to other locations in memory where the data is stored primitively. Primitive types include numeric types like integers, floats, and doubles. Non-numeric primitive types in Java help to store data like Booleans, bytes, and characters.

In many other languages (like C), Strings are simply arrays of character data. Java, however, has its own String class, which abstracts much of the underlying complexities. Therefore, a String

data type in Java is a reference type.

Structure

This chapter will cover the following programs:

- Exploring instance and static methods
- Program 3.1: Reverse a string
- Program 3.2: Check if a primitive character is uppercase
- Program 3.3: Check if a reference character is uppercase
- Program 3.4: Format a phone number
- Program 3.5: Check if a state code is valid
- Program 3.6: Check if a state code is valid using regular expression
- Program 3.7: Check if a string is a palindrome
- Program 3.8: Parse and format hockey team data

Objectives

The primary goal of this chapter is for readers to gain a working understanding of Character and String data types. This includes challenges with reading, storing, and changing text data, as well as how to leverage many of the Character and String class methods to achieve desired results. Readers should also look to gain an introductory understanding of concepts like character encoding and regular expressions, as well as how they can be used to solve complex problems.

Exploring instance and static methods

Reference types have two kinds of methods available for developers to use: static methods and instance methods. Static methods can be called directly, without requiring an object of the class to be first instantiated. The **isUpperCase()** method from the **Character** class is an example of a static method:

```
return Character.isUpperCase('a');
```

On the other hand, to use an instance method, an object of that class must first be instantiated. The **charAt()** method from the **String** class is an example of an instance method:

```
String message = "Hello!";
char letterE = message.charAt(1);
```

Note: Many Java primitive types also have matching reference types, which offer additional, delivered methods to help process the underlying data.

We will discuss and use both instance and static methods in this chapter.

Program 3.1: Reverse a string

Take a string as input and return it with its characters in reverse order.

Note: This problem is a common technical interview question for junior-level developer positions.

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **reverseString**, which accepts a **String** variable named **inputStr**. This method should also have a return type of **String**.
2. Instantiate a new variable named **reversed** of type **StringBuilder**.
3. Create a for loop with a counter named **arrayIndex**. Initialize **arrayIndex** to the number of characters present in **inputStr**, and decrement from back to front.
4. Use the **charAt()** **String** method to access the current character denoted by **arrayIndex**. Add this character to the **reversed StringBuilder** object with the **append()** method.
5. Once the loop has completed, use the **toString()** method to return the **String** value of **reversed** to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 3.1*:

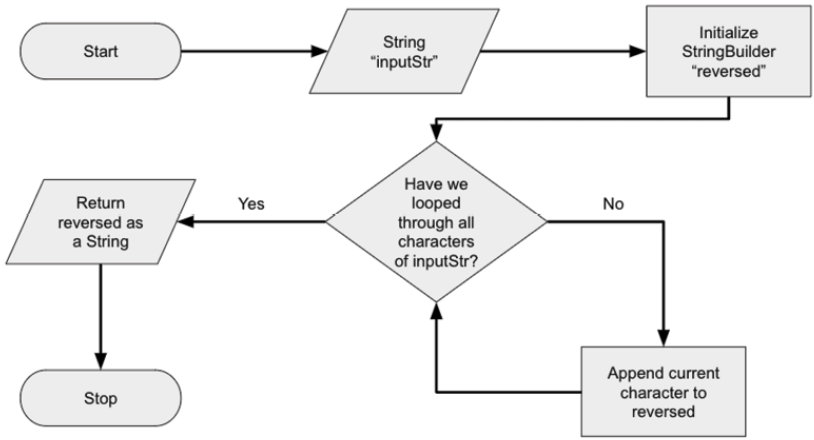

***Figure 3.1**: Flowchart describing how to reverse a string*

Java code

The Java code for *Program 3.1* is as follows:

```
private static String reverseString(String inputStr) {
    StringBuilder reversed = new StringBuilder();
    for (int arrayIndex = inputStr.length(); arrayIndex > 0;
arrayIndex--) {
        reversed.append(inputStr.charAt(arrayIndex - 1));
    }
    return reversed.toString();
}
```

Code explanation

To solve this problem, we will start by creating a new method named **reverseString**. This method will accept one **String** parameter named **inputStr**, and will also return a value of type **String**. Inside our new method, we will instantiate a new **StringBuilder** object named **reversed**.

Note: A StringBuilder is a special kind of class that lets us work with strings in a mutable fashion.

Next, we will use a for loop and initialize its counter variable **arrayIndex** to the number of characters in **inputStr**. This loop will then decrement **arrayIndex** each time through. Inside the loop, we will take the current character (designated by **arrayIndex** - 1) of **inputStr** using the **charAt()** method and **append()** it to **reversed**. Once the loop is complete, we will return the **String** value of **reversed**, using the **toString()** method.

Program 3.2: Check if a primitive character is uppercase

For this program, we will simply write a method to accept a primitive character type and return a Boolean indicating whether it is uppercase (true) or lowercase (false).

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **isPrimitiveUppercase**, which accepts a character variable (primitive type) named **inputChar**. This method should return a Boolean value.
2. First, we will initialize a new integer variable named **asciiCode**, and set it to the result of converting **inputChar** to an integer.
3. Next, we will check whether the value of **asciiCode** is greater than 64 and less than 91. If it is, we will return a value of true to the calling program.
4. Otherwise, we will return a value of false to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 3.2:*

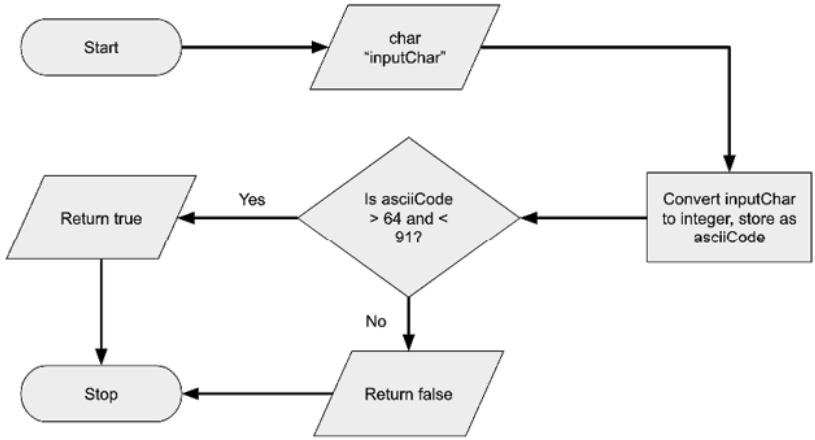

***Figure 3.2**: Flowchart describing how to check if a given primitive character is uppercase*

Java code

The Java code for *Program 3.2* is as follows:

```
private static boolean isPrimitiveUppercase(char inputChar) {
    int asciiCode = (int) inputChar;
    if (asciiCode > 64 && asciiCode < 91) {
        return true;
    }
    return false;
}
```

Code explanation

To solve this problem, we will write a new method named **isPrimitiveUppercase**. It will accept a (primitive) character variable named **inputChar**. We will then instantiate a new integer variable named **asciiCode**, and set it to the result of **(int) inputChar**. This line of code runs an implicit conversion between the character and integer primitive types. The result is that the underlying ASCII/UTF code for the character value is returned (as an integer).

Note: American Standard Code for Information Interchange (ASCII) was the character encoding standard for early computers. Today's computers use the Universal Transformation Format (UTF) for

character encoding. This includes (Ploetz 2024) the ASCII code set (0-255) as well as more than 1 million character encodings to include symbols from many different languages.

Next, we will check if the value of **asciiCode** is both greater than 64 and less than 91. Valid, uppercase characters have a value of 65 (letter A) to 90 (letter Z). If the value of **asciiCode** is within that range, we know that the character was indeed uppercase, so we will return true to the calling program. Otherwise, should the program flow progress beyond the if construct, we will simply return false.

Program 3.3: Check if a reference character is uppercase

For this program, we will simply write a method to accept a reference character type and return a Boolean indicating whether it is uppercase (true) or lowercase (false).

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **isReferenceUppercase**, which accepts **character** variable (reference type) named **inputChar**. This method should return a Boolean value.
2. We will call the static method **isUpperCase()** from the **Character** class, while passing **inputChar** as the lone parameter. We will return the result of this method to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 3.3*:

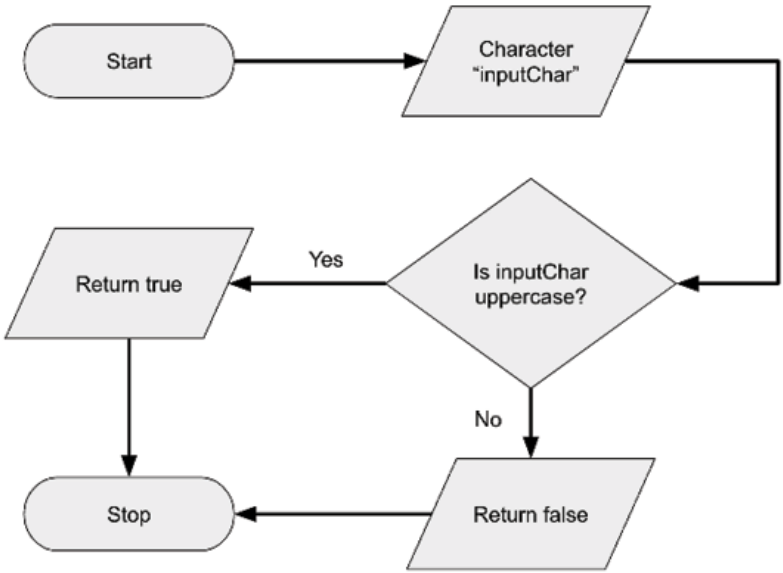

***Figure 3.3**: Flowchart describing how to check if a given reference character is uppercase*

Java code

The Java code for *Program 3.3* is as follows:

```
private static boolean isReferenceUppercase(Character inputChar) {
    return Character.isUpperCase(inputChar);
}
```

Code explanation

To solve this problem, we will write a new method named **isReferenceUppercase**. It will accept a (reference) **Character** variable named **inputChar**. We will use the **isUpperCase()** static method from the **Character** class. We will pass **inputChar** to that method. We will return that result to the calling program.

This program helps to highlight some of the benefits of using reference data types. Both **isReferenceUppercase** and **isPrimitiveUppercase** accomplish the same task. While this example has shown how powerful the additional helper methods can be, it is important to remember that (*Baeldung* 2024) primitive types use a smaller memory footprint and provide higher-performing data

access.

Program 3.4: Format a phone number

Take a 10-digit phone number from user input and return it with consistent formatting.

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **isReferenceUppercase**, which accepts a phone number stored in a **String** variable named **phoneNumber**.
2. Remove all leading and trailing spaces from **phoneNumber**, and store the result in a new **String** named **number**.
3. Remove additional characters from **number**:
    a. Dashes b. Dots/periods
    c. Parens d. Spaces
4. If the remaining character length of **number** is 10:
    a. Instantiate a new **String** named **areaCode** with the first 3 characters of the number.
    b. Instantiate a new **String** named **prefixCode** with next 3 characters.
    c. Instantiate a new **String** named **lineCode** with the remaining 4 characters.
    d. Instantiate a new **StringBuilder** named **returnVal** to assemble and format the **areaCode**, **prefixCode**, and **lineCode**.
    e. Return the **String** value of **returnVal** to the calling program.
5. If the remaining character length of the number is not equal to 10, then return a message stating, **INVALID NUMBER, 10 digits only!**

Flowchart

The flowchart for this problem can be seen in *Figure 3.4*:

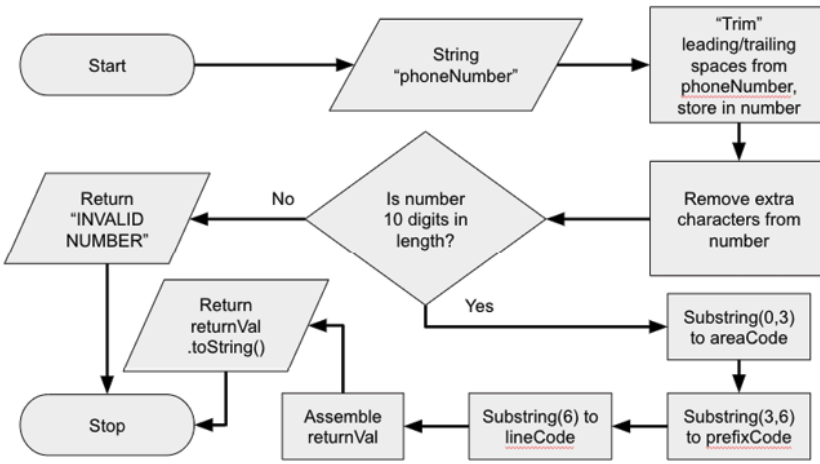

***Figure 3.4**: Flowchart describing how to process a 10-digit phone number*

Java code

The Java code for *Program 3.4* is as follows:

```
private static String formatPhoneNumber(String phoneNumber) {
    // remove leading and trailing spaces
    String number = phoneNumber.trim();
    // remove extra characters ()-.
     number = number.replaceAll("\\D", "");
    // number should be 10 characters, otherwise number is invalid
    if (number.length() == 10) {
        // AREACODE-PREFIX-LINENUMBER
        String areaCode = number.substring(0,3);
        String prefixCode = number.substring(3,6);
        String lineCode = number.substring(6);
        StringBuilder returnVal = new StringBuilder("(");
        returnVal.append(areaCode);
        returnVal.append(")");
```

```
        returnVal.append(prefixCode);
        returnVal.append("-");
        returnVal.append(lineCode);
        return returnVal.toString();
    } else {
        return "INVALID NUMBER, 10 digits only!";
    }
}
```

Code explanation

To solve this problem, we will create a new method named **formatPhoneNumber**. It will accept a **String** parameter named **phoneNumber**.

As user input can be unpredictable, we will begin by removing unnecessary characters from **phoneNumber**. First, we will instantiate a new **String** variable named **number**, and initialize it with the result of the **String** method **trim()** to remove any leading and trailing spaces. We will then call the **String** method **replaceAll("\\D","")** to remove any non-numeric characters that users tend to add into phone number inputs, like dashes, dots, spaces, and parentheses.

Before proceeding any further, we will verify that **number** has 10 digits by using the **String** method **length()**. If the result of **number.length()** is not 10, we will immediately return a message of **"INVALID NUMBER, 10 digits only!"** to the calling program.

We will then take our 10-digit number and break it up into three separate parts: area code, prefix code, and line code. To do this, we will use the **String** method **substring()**. The **substring()** method returns a smaller string designated by its two integer parameters: the beginning character index and the ending character index.

Using this approach, we will instantiate three **String** variables named **areaCode**, **prefixCode**, and **lineCode**. The **substring(0,3)** method call will store the numbers at **String** index positions 0, 1, and 2 into **areaCode**. The **substring(3,6)** method call will store the numbers at **String** index positions 3, 4, and 5 into the **prefixCode**.

Finally, the **substring(6)** method call will store all of the remaining numbers starting at **String** index position 6 into **lineCode**.

Finally, we will instantiate a new **StringBuilder** named **returnVal** and append **areaCode**, **prefixCode**, and **lineCode** with our own, consistent formatting. We will then return the result of **returnVal**'s **toString()** method back to the calling program.

Program 3.5: Check if a state code is valid

Take a 2-character (United States) state code and return whether it is valid (true) or invalid (false).

Algorithm

To solve this problem, we will take the following steps:

1. Instantiate a private, final integer variable named **STATE_LENGTH** with a value of 2.
2. Create a new method named **isStateCodeValid**, which accepts a **String** parameter named **state** and returns a Boolean value.
3. Initialize a Boolean variable named **returnVal** to a value of **false**.
4. If the **length()** of **state** is equal to **STATE_LENGTH**:
    a. Iterate through each character in the **state String**.
    b. If the character returns a value of **true** from the **isUpperCase()** method, then set **returnVal** to a value of **true**. Otherwise, set **returnVal** to **false** and break out of the loop.
5. If the **length()** of **state** is not equal to **STATE_LENGTH**, then return the current value of **returnVal** (which should be false) to the calling program.
6. Return the current value of **returnVal** to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 3.5*:

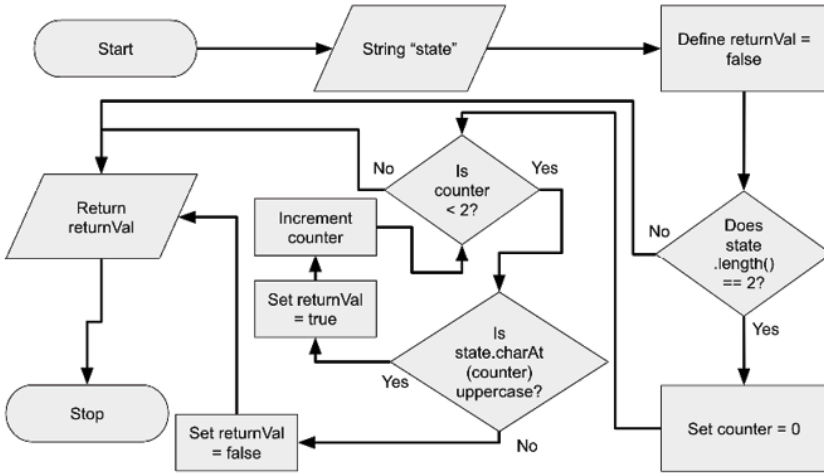

***Figure 3.5**: Flowchart describing how to validate a 2-character state code*

Java code

The Java code for *Program 3.5* is as follows:

```
private static final int STATE_LENGTH = 2;
private static boolean isStateCodeValid(String state) {
    boolean returnVal = false;
    if (state.length() == STATE_LENGTH) {
        for (int counter = 0; counter < STATE_LENGTH; counter++)
{
            if (Character.isUpperCase(state.charAt(counter))) {
                returnVal = true;
            } else {
                returnVal = false;
                break;
            }
        }
    }
    return returnVal;
}
```

Code explanation

First and foremost, it is important to understand that we have two very simple requirements. All United States state codes must be:

- Uppercased
- 2-characters in length

To be successful, our code should accurately reflect and adhere to these requirements.

To solve this problem, we will define an integer constant named **STATE_LENGTH** with a value of 2. We will then create a new method named **isStateCodeValid**. It will return the value of type Boolean and accept a parameter of type **String** named **state**. We will also instantiate a Boolean variable named **returnVal** with a value of **false**.

Note: The practice of using constants improves performance, as it prevents variables from having to be redeclared. It also improves code readability.

Next, we will check the number of characters in the **state String** using the **length()** method. If the result of **state.length()** is equal to **STATE_LENGTH (2)**, we will create a for loop to iterate over an integer variable named **counter**. We will use the value of **counter** as an index to access each character in the **state** string, using the **charAt()** method.

If the current character is indeed uppercased, we will set **returnVal** to a value of **true**. If the current character is not uppercased, we will set **returnVal** to a value of **false and** break out of the loop. Since both characters need to be uppercased, we can (and should) short-circuit the loop if the first character is lowercased, as there is no need to check the second character. However, if the first character is uppercased, we still need to make sure that the second character is also uppercased.

Finally, we will return the value of **returnVal** to the calling program. This will happen immediately if the initial check of **state.length()** is anything other than 2. In that case, there is no need to perform any further verifications.

Program 3.6: Check if a state code is valid using regular expression

Take a 2-character (United States) state code and (using a regular expression) return whether it is valid (true) or invalid (false).

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **isStateCodeValidRegex**, which accepts a **String** parameter named **state** and returns a Boolean value.
2. Define a new regular expression pattern named **statePattern**, using the **compile()** static method from the **java.util.regex.Pattern** class. The pattern should be the String value of **[A-Z]{2}**.
3. Define a new regular expression matcher named **stateMatcher**. Initialize **stateMatcher** with the result of the **statePattern.matcher()** method, while passing **state** as a parameter.
4. Return the result of the **stateMatcher.matches()** method to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 3.6*:

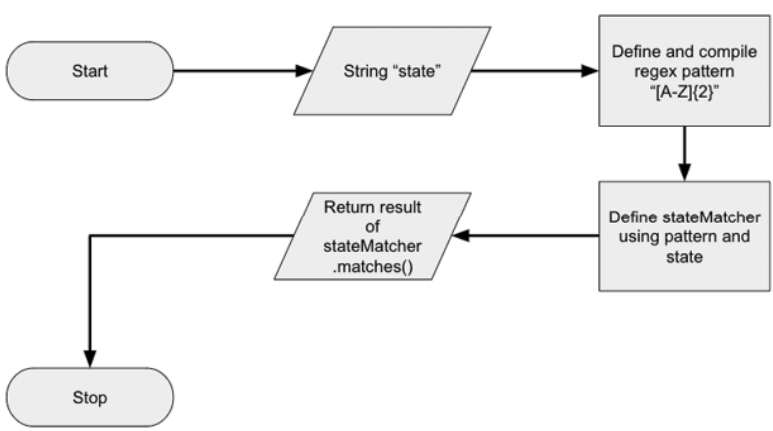

***Figure 3.6**: Flowchart describing how to validate a 2-character state code using a regular expression*

Java code

The Java code for *Program 3.6* is as follows:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
private static boolean isStateCodeValidRegex(String state) {
    Pattern statePattern = Pattern.compile("[A-Z]{2}");
    Matcher stateMatcher = statePattern.matcher(state);
    return stateMatcher.matches();
}
```

Code explanation

Before building this method, our class will need to import two Java library classes: **java.util.regex.Pattern** and **java.util.regex.Matcher**. This will allow us to invoke the core methods that make Java regular expressions possible.

To solve this problem, we will create a new method named **isStateCodeValidRegex**. It will accept a **String** parameter named state and return a Boolean value.

Inside the method, we will instantiate a new **Pattern** object named **statePattern**. This variable should be initialized to the result of the **Pattern.compile()** method, that takes the **String** value of **[A-Z]{2}**. This indicates that we want to match on two (2) uppercase letters (A-Z) only.

Next, we will instantiate a new **Matcher** object named **stateMatcher**. This should be initialized with the result of the **statePattern.matcher()** method, which takes the **state String**. Finally, we simply need to return the Boolean result of the **stateMatcher.matches()** method.

While this application of regular expressions may seem like the simpler path, it will perform more slowly than the iterative approach used in *Program 3.5*. Regular expressions are a great tool that can be used to solve complex problems. The problem of

validating a 2-character state code presents an easy way to show how regular expressions could be used. But this problem is already simple enough that it does not offset the performance cost presented by using regular expressions.

It is also worth noting that the part that performs the slowest is the pattern compilation step. The preferred way to use a regular expression in Java would be to execute the **Pattern.compile()** method once (probably in the calling program or method), pass the pattern as a parameter, and then match against it as many times as are necessary.

Program 3.7: Check if a string is a palindrome

Take a **String** and determine if it can be expressed the same when its characters are in reverse order. We can test it with **String** data like this:

```
COBOL
madam
rotator
```

From these tests, we should see output similar to this:

```
Is COBOL a palindrome? false
Is madam a palindrome? true
Is rotator a palindrome? true
```

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **isPalindrome**, which returns a Boolean and accepts a **String** named **word**.
2. Use a for loop to iterate an integer named **index** if it is less than the length of **word** divided by 2.
3. Inside the loop, instantiate a character named **fromFront** with the value of running the **charAt()** method on the **word String** with a parameter of **index**.
4. Instantiate a second character named **fromBack** with the

value of running the **charAt()** method on the **word String** with a parameter of **word.length() – 1 - index**.

5. If these **fromFront** and **fromBack** are not equal, return **false**.
6. If the loop has completed, return **true**.

Flowchart

The flowchart for the **isPalindrome()** method can be seen in *Figure 3.7*:

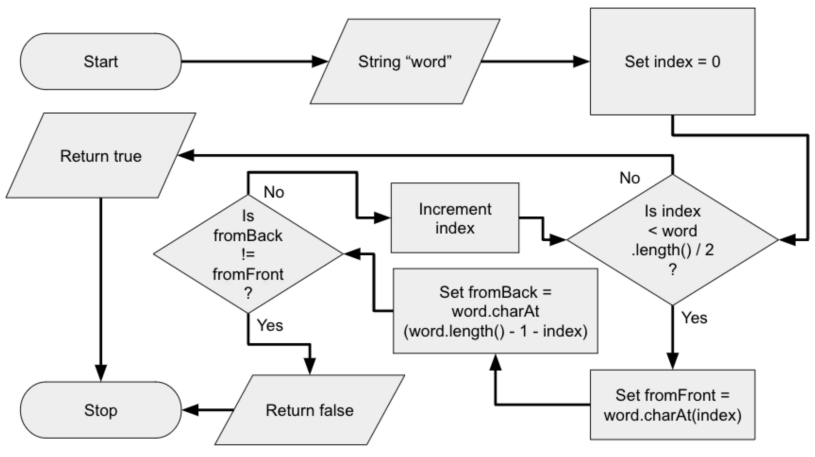

***Figure 3.7**: Flowchart describing how to tell if a string is a palindrome*

Java code

The Java code for *Program 3.7* is as follows:

```
private static boolean isPalindrome(String word) {
    for (int index = 0; index < (word.length() / 2); index++) {
        char fromFront = word.charAt(index);
        char fromBack = word.charAt(word.length() - 1 - index);
        if (fromFront != fromBack) {
            return false;
        }
    }
```

```
    return true;
}
```

Code explanation

To solve this problem, we will create a new method named **isPalindrome**. This method will return a Boolean and accept a **String** parameter named **word**.

Note: To be clear, a palindrome is a word that is spelled the same forward as it is backward.

Inside the method, we will build a for loop. This loop will instantiate an integer **index**, and increment the **index** if the **index** is less than the result of **word.length()** divided by 2. After all, if the front half of the word looks the same as the back half, then we only need to iterate over half of the letters in the word.

Within the for loop, we will first define a character variable named **fromFront**, and assign it the value of the current character as denoted by **index**. We can do this by using the **charAt()** method on **word**, with a parameter of **index**. Likewise, we will also define a character variable named **fromBack**. We will assign it to the character positioned at the value of **index** from the end of the **String**, by calling the **charAt()** method on **word**, with a parameter of **word.length() - 1 - index**. Next, we will perform an if check on **fromFront** and **fromBack** for equality. If they are not equal, we return a value of **false** to the calling program.

Should the for loop complete without returning a value of **false**, we will then return a value of **true** to the calling program.

Program 3.8: Parse and format hockey team data

Take a pipe-delimited row of hockey team data, parse out the team's name and location, and display them appropriately. We will test our program with data that looks like this:

```
"city: St. Paul, MN | team: Wild | location: Minnesota | colors: Green, White"

"city: Toronto, ON | team: Maple Leafs | location: Toronto | colors: Blue, White";
```

"city: Sunrise, FL | team: Panthers | location: Florida | colors: Red, Blue, Brown";

"city: Dallas, TX | team: Stars | location: Dallas | colors: Green, Black";

"city: Pittsburgh, PA | team: Penguins | location: Pittsburgh | colors: Yellow, White, Black";

When successful, our output should look like this:

Minnesota Wild

Toronto Maple Leafs

Florida Panthers

Dallas Stars

Pittsburgh Penguins

Algorithm

To solve this problem, we will take the following steps:

1. Create a new method named **getValue**, which accepts a **String** named **kvPair**.
    a. Inside this method, instantiate a new integer named **colonPosition** with the value returned from running the **indexOf()** method on the **kvPair String**, with a parameter of **":"**.
    b. Next, return the result of the **substring()** method on the **kvPair String**, with a parameter of **colonPosition + 2**. Also, add a call on the return value to the **trim() String** method, to remove any extra trailing or leading spaces from the result.
2. Create a new method named **getFullTeamName**, which accepts a pipe-delimited **String** of key/value pairs named **teamData**.
    a. Inside this method, define the following variables:
        i. A new **StringBuilder** named **returnVal**.
        ii. A **String** array named **teamDataArray**, initialized to the result of running the **split() String** method on **teamData** with a parameter of **[|]**.

iii. A **StringBuilder** named **name**.

iv. A **StringBuilder** named **location**.

b. Create a for loop which instantiates an integer named index to 0 and runs if index is less than **teamDataArray.length**.

i. Inside the for loop, check if the current item referenced by **index** in **teamDataArray** contains the **String "team:"**. If it does, pass the item to our **getValue()** method and append the result to the **name** variable.

ii. Check if the current item referenced by **index** in **teamDataArray** contains the **String "location:"**. If it does, pass the item to our **getValue()** method and append the result to the **location** variable.

c. Append **location**, a single space, and **name** to **returnVal**.

d. Return the value of running the **.toString()** method on **returnVal** to the calling program.

Flowchart

The flowchart for the **getValue()** method can be seen in *Figure 3.8*:

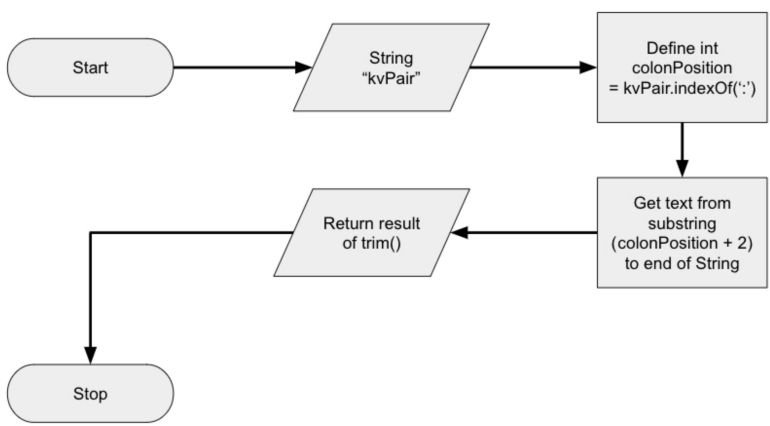
Start
String "kvPair"
Define int colonPosition = kvPair.indexOf(':')
Get text from substring (colonPosition + 2) to end of String
Return result of trim()
Stop

***Figure 3.8**: Flowchart describing how the getValue() method works*

The flowchart for the **getFullTeamName()** method can be seen in

Figure 3.9:

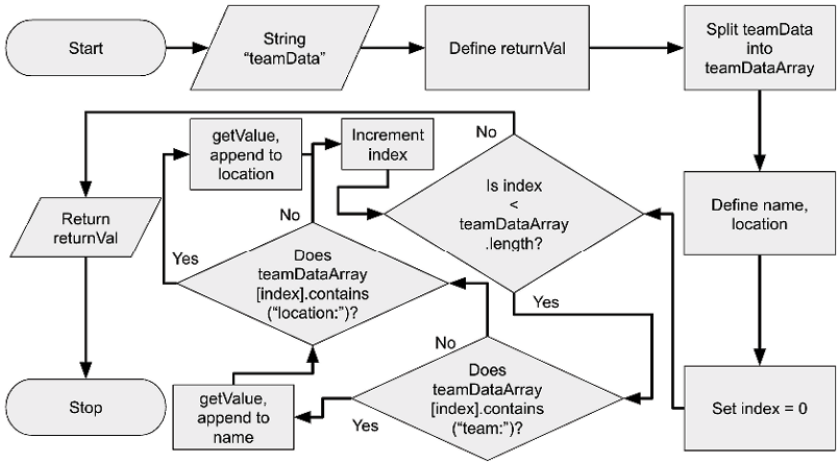

***Figure 3.9**: Flowchart describing how to process hockey team data*

Java code

The Java code for *Program 3.8* is as follows:

```
private static String getValue(String property) {
    int colonPosition = property.indexOf(':');
    return property.substring(colonPosition + 2).trim();
}
private static String getFullTeamName(String teamData) {
    StringBuilder returnVal = new StringBuilder();
    String[] teamDataArray = teamData.split("[|]");
    StringBuilder name = new StringBuilder();
    StringBuilder location = new StringBuilder();
    for (int index = 0; index < teamDataArray.length; index++) {
        if (teamDataArray[index].contains("team:")) {
            name.append(getValue(teamDataArray[index]));
        }
        if (teamDataArray[index].contains("location:")) {
            location.append(getValue(teamDataArray[index]));
```

```
        }
    }
    returnVal.append(location);
    returnVal.append(" ");
    returnVal.append(name);
    return returnVal.toString();
}
```

Code explanation

To solve this problem, we will create two methods. The first will be named **getValue**, which will return a **String**, and accept a **String** parameter named **kvPair** (representing a **String** containing a key and a value paired together).

The separator in **kvPair** will be a colon (**:**). Therefore, we will define a new integer variable named **colonPosition**, and set it to the result of the **indexOf() String** method when run on the **kvPair** variable with a parameter of **":"**. To access the property value only, we will need to run the **substring() String** method on the **kvPair** variable with a parameter of **colonPosition + 2**. Calling **substring()** with one parameter will use that parameter as a starting point and copy all characters until it reaches the end of the **String**. We will then also call the **trim() String** method on this result to remove any extra spaces before returning it to the calling method.

With our **getValue()** method written, we can now focus on writing our primary method for this program, named **getFullTeamName**. The **getFullTeamName** method will return a **String** value and accept a **String** parameter named **teamData**.

To successfully process all the variants of **teamData**, we will need to define a few variables. First, we will define a **StringBuilder** named **returnVal**, which will be used to assemble and return our results. Next, we will define a **String** array named **teamDataArray**, and initialize it with the result of running the **split()** method on **teamData** with a parameter of **[|]**. This will split the text in **teamData** as delimited by the pipe | character.

The last two variables will be defined as **StringBuilders** named

name and **location**. We will use these variables to append our relevant data as we process it and ultimately build them into **returnVal**.

Note: The use of StringBuilders for name and location gets us around having to define them as empty strings. This leads to more efficient use of our underlying resources, as strings are immutable (cannot be changed) in Java.

With our variables defined, we can now build a for loop. This loop will instantiate a variable named **index** and run as long as the value of **index** is less than the number of items (**length**) in **teamDataArray**.

As we increment over each key/value pair stored in **teamDataArray**, we will be looking for the team and location keys. We will first check to see if the current item (denoted by **teamDataArray[index]**) has data for the team property, by using the **contains() String** method while passing **"team:"** as a parameter. If it does, we will send the **teamDataArray** item to our **getValue()** method and append the result to **name**. Likewise, we will then check the current item for the location by using the **contains() String** method while passing **"location:"** as a parameter. If it does, we will send the **teamDataArray** item to our **getValue()** method and append the result to **location**.

With the for loop complete, we will append location, a single space, and name to **returnVal**. We will then call the **toString()** method on **returnVal**, and return that value to the calling program.

Conclusion

In this chapter, we covered characters and strings and learned how to solve problems using their provided methods. We used these methods in approaches to problems that had us working on reading, writing, and applying custom processing to text data.

We looked at the Character class and learned how to check for case. We also learned that we could cast a character to an integer, and that the integer returned reveals its underlying UTF code.

The programs in this chapter also made good use of the String class. We learned how to split strings, trim strings, create substrings, and

iterate through the underlying character index positions in a string. We also had a brief introduction to regular expressions and learned about the tradeoffs that accompany them.

In the next chapter, we will build an understanding of basic computer arithmetic. We will convey this transfer of fundamental knowledge by writing some basic operational methods of our own.

Points to remember

Have a look at some key points to remember:

- Many Java primitive data types also have matching reference types. Example: The character primitive has a corresponding character reference type.
- ASCII was an earlier, widely used character encoding standard. Today's computers use UTF-8 for character encoding. In Java, strings are represented in UTF-16.
- Under the hood, Java strings are immutable (cannot be changed or updated).
- Casting a character to an integer reveals its underlying ASCII/UTF code.
- Regular expressions are slow but can be useful for problems that require complex pattern matching.
- If you are going to use a regular expression, try to compile your pattern only once.
- The substring() method can be used to copy part of a string into another string variable.
- The trim() method will remove both leading and trailing spaces from a string value.
- In Java, defining an empty string is considered to be a bad practice.
- A StringBuilder is a good option for string values that need to be frequently updated or assembled at process time.

Questions

Answer the following questions to check your understanding:

1. Describe how you could iterate through a loop backwards. What would be different versus a standard, forward iterating loop?
2. How would Program 3.2 have to change to determine if a primitive character is lowercase, instead of uppercase?
3. Give an example of how to replace a question backslash (\) with a forward slash (/) in a string.
4. Given the string of ATARI 2600 CARTRIDGE GAMES how could you initialize a new String variable named model to the 2600 part of the String?
5. Which static string method allows you to view how many characters are in a String?
6. Define a regular expression pattern to match on a string of 5 lowercase characters.
7. Given the string “console: Sega Genesis, game: NHL 94, year: 1993”, write a method to retrieve the value of the game property.
8. Examine the code inside the for loop in Program 3.7. Is there a way that we could get it to perform more efficiently?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Section 2:
Intermediate Problems

Chapter 4

Arithmetic Solutions

Introduction

Arithmetic is the core of computing. Everything we do, even in third-generation languages like Java, eventually breaks down into basic arithmetic operations. Therefore, understanding these operations and knowing how to leverage them to solve problems is an essential skill for any developer to have.

In this chapter, we will cover some basic arithmetic operations and even write some purpose-built methods of our own. We will then show how these methods can be used to solve complex problems.

Structure

This chapter will cover the following programs:

- Program 4.1: Multiply two numbers
- Program 4.2: Subtract two numbers
- Program 4.3: Divide two numbers
- Program 4.4: Check if a number is a power of two

Objectives

The primary objective of this chapter is to help readers build an understanding of basic computer arithmetic. We will convey this transfer of fundamental knowledge by writing some basic operational methods of our own. These methods are important to understand, as we will use some of them to solve other problems later in the chapter.

The difficulty of our program flow will also increase in this chapter, as we introduce the concept of recursion. A recursive method is a method that calls itself and can be used to solve uniquely complex problems. As recursion is a valuable concept for any developer to understand, we will demonstrate its use in this chapter.

Toward the end of this chapter, we will build methods designed to work with binary components of integers. We will discuss the *power of two* numbers, why they are important to us, and demonstrate how to use them.

Note: The test code given at the beginning of each problem should be added to a new class file, inside of a standard public static void main(String[] args) method. Failure to do so may result in an unnamed class exception. For our purposes, a class named Chapter4 will work just fine.

Program 4.1: Multiply two numbers

Take two numbers and compute their product (result of a multiplication operation) without using the multiplication (*) operator.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new method named **multiply** that accepts two integers named **numX** and **numY**, and returns an integer.
2. First, check if the value of either **numX** or **numY** is equal to 0. If so, then return 0 to the calling program.

3. Next, we will instantiate two variables:
    a. A Boolean named **isYNegative**, with a value of **false**.
    b. An integer named **returnVal**, with a value of 0.
4. Check if **numY** is less than 0. If so, set **isYNegative** to a value of **true**.
5. If **isYNegative** has a value of **true**, then:
    a. Write a for loop which decrements an integer named **counter** as long as it is greater than **numY**, and decrements **returnVal** by the value of **numX** on each iteration.
6. Otherwise, if **isYNegative** has a value of **false**, then:
    a. Write a for loop which increments an integer named **counter** as long as it is less than **numY**, and increments **returnVal** by the value of **numX** on each iteration.
    b. Return the value of **returnVal** to the calling program.

Flowchart

The flowchart for this problem can be seen in *Figure 4.1*:

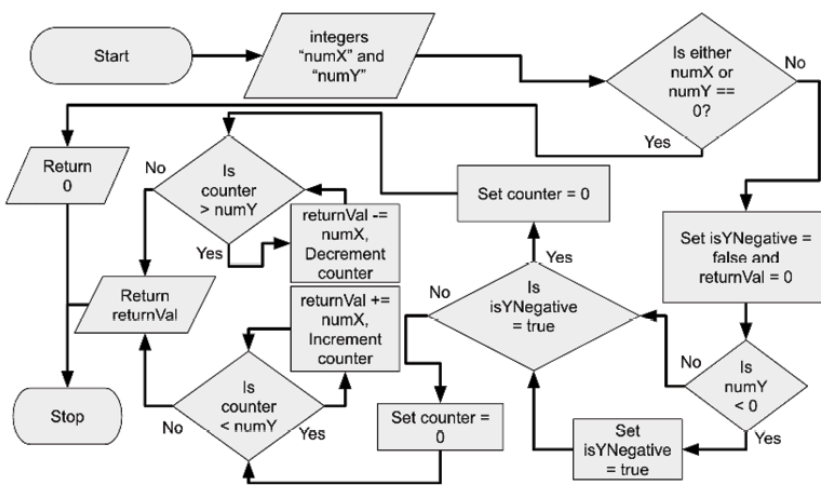

***Figure 4.1**: Flowchart describing how to multiply two numbers together*

Java code

The Java code for *program 4.1* is specified as follows:

```
public static int multiply(int numX, int numY) {
    // if either numX or numY are 0, then short-circuit the method
    if (numX == 0 || numY == 0) {
        return 0;
    }
    boolean isYNegative = false;
    int returnVal = 0;
    if (numY < 0) {
        isYNegative = true;
    }
    if (isYNegative) {
        for (int counter = 0; counter > numY; counter--) {
            returnVal -= numX;
        }
    } else {
        // Y is positive
        for (int counter = 0; counter < numY; counter++) {
            returnVal += numX;
        }
    }
    return returnVal;
}
```

Code explanation

Before solving this problem, it is important to discuss *why* we are solving this problem. After all, why should we build our own multiplication method when Java and all other programming languages have a multiplication operator?

The answer is two-fold. First, by breaking down the multiplication operation, we are building further understanding of how it works. Secondly, this is actually a real-world problem that is solved by

lower-level code for different CPU architectures, like **Reduced Instruction Set Computer** (**RISC**) machines. Machines based on RISC architectures only have a bare minimum of delivered machine code methods built into the processor itself and thus rely on low-level software-based methods for additional functionality.

Note: Most consumer-grade laptops and desktop computers are based on Complex Instruction Set Computer (CISC) processors. However, it is becoming more common for some developers to use laptops equipped with advanced RISC machine (ARM) processors.

To begin, we will write a new method named **multiply**. It will accept two integer parameters named **numX** and **numY**, and it will return an integer.

We will first check whether either **numX** or **numY** is equal to 0. After all, the product of any two numbers when one of them is 0 is 0. If this is the case, we can simply return 0 to the calling program, thus short-circuiting the rest of the method.

Next, we will define two variables. The first will be a Boolean named **isYNegative**, and it will have a value of **false**. The other will be an integer named **returnVal**, and it will have a value of 0. Likewise, we will also quickly check if **numY** is less than 0 and then reassign **isYNegative** accordingly.

Then, based on the value of **isYNegative**, we will execute one of two for loops. If **isYNegative** has a value of true, we will implement a for loop that will count backwards (decrement) from 0 to **numY**, decrementing **returnVal** by the value of **numX** on each iteration. Conversely, if **isYNegative** has a value of false, we will implement a for loop that will increment from 0 to **numY**, incrementing **returnVal** by the value of **numX** on each iteration. When the loop completes, we will return the value of **returnVal** to the calling program.

Program 4.2: Subtract two numbers

Take two numbers, subtract the second from the first, and return the result without using the delivered subtraction (-) operator.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new method named **applyNegative** that accepts a single integer named **number** and returns an integer. Inside this method:
    a. Using the **multiply()** method that we built in *Program 4.1*, return the result of passing **number** and a value of **-1** as parameters to the calling method.
2. Write a new method named **subtract** that accepts two parameters, both integers named **numX** and **numY**, respectively, and returns an integer. Inside this method:
    a. Get the result of **numX** plus (+) the result of the **applyNegative()** method with **numY** passed as a parameter, and return it to the calling program.

Flowchart

The flowcharts for this problem can be seen in *Figures 4.2* and *4.3*:

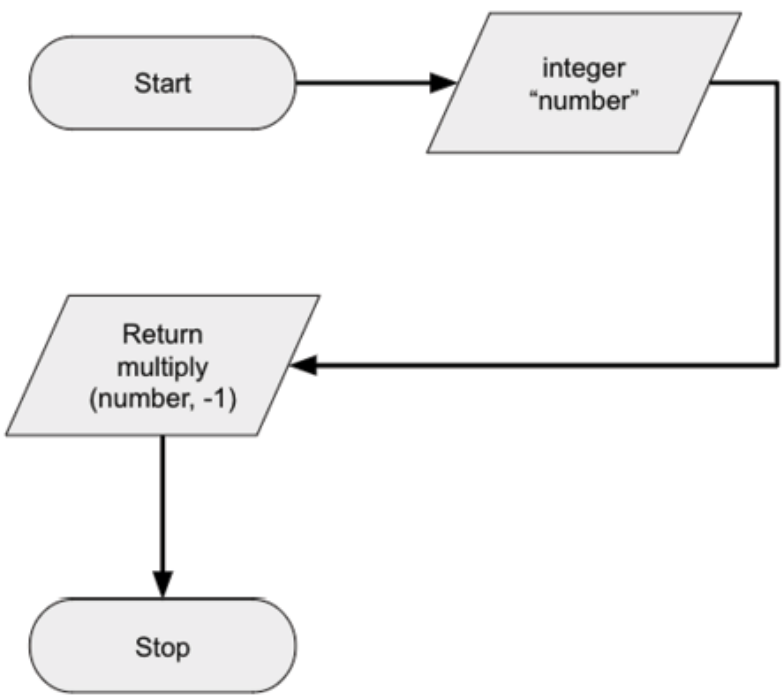

***Figure 4.2**: Flowchart describing the applyNegative() method*

The following figure shows a simple flow for computing a result by combining two integers, applying a negative transformation before returning the final value:

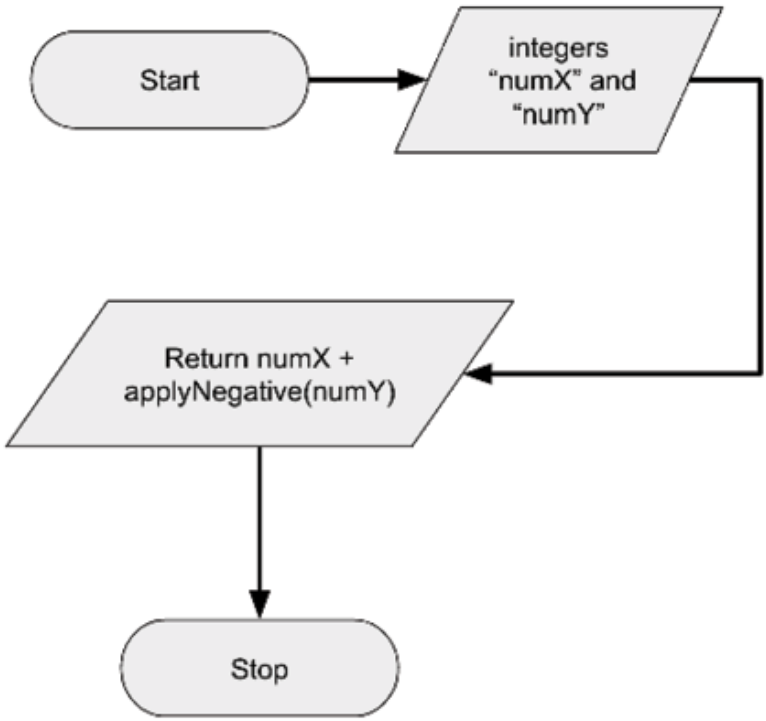

***Figure 4.3**: Flowchart describing the subtract() method*

Java code

The Java code for *Program 4.2* is specified as follows:

```
Private static int applyNegative(int number) {
    return multiply(number,-1);
}
private static int subtract(int numX, int numY) {
    return numX + applyNegative(numY);
}
```

Code explanation

The first thing we need to do here is to write a new method named **applyNegative**. This method should accept an integer named **number** as a parameter and return an integer. This method has the simple, yet crucial function of returning the mathematical negative of **number**.

Note: Remember that the negative of a negative is positive. Therefore, negative numbers sent to the applyNegative() method will be returned as positives.

Next, we can write our new **subtract** method. This method will accept two integer parameters named **numX** and **numY**, respectively, and return an integer. This method will simply return the result of **numX** + **makeNegative(numY)**.

While these two methods seem very simple, they will be used by other methods in this chapter. Now, we certainly could have built the logic from the **makeNegative()** method into our **subtract()** method, but solving problems is often easier once you break them into smaller problems and build a method for each of those small problems. This will become apparent in the remainder of the chapter.

Program 4.3: Divide two numbers

Take two numbers, a dividend and a divisor, and compute their quotient (result of division) without using the division (/) operator.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new method named **abs**. This method will accept one integer parameter named **number**, and it will return an integer value. The purpose of this method will be to compute and return the absolute value of **number**. Inside this method:
    a. Check if **number** is < 0. If it is less than 0, then return the result of the **applyNegative()** method with **number** as the parameter to the calling method.
    b. Otherwise, return the value of **number**.

c. Write a new method named **divide** that accepts two integers named **dividend** and **divisor**, and returns an integer. Inside this method, we will:

d. Define a Boolean named **resultShouldBeNegative** with a value of **true**.

i. Check whether **divisor** and **dividend** are either:

ii. Both are less than 0.

e. Both greater than 0.

f. If either one of those scenarios is true, then set **resultShouldBeNegative** to a value of false.

g. Define an integer variable named **absoluteDividend** and instantiate it with the result of the **abs()** method using **dividend** as a parameter.

h. Define an integer variable named **absoluteDivisor** and instantiate it with the result of the **abs()** method using **divisor** as a parameter.

i. Define two additional integer variables named **total** and **quotient**, both with a value of 0.

i. Build a while loop that runs as long as the sum of **total** + **absoluteDivisor** is less than or equal to **absoluteDividend**. Inside the loop:

ii. Set total to the sum of **total** + **absoluteDivisor**.

j. Increment **quotient**.

k. Once the loop is complete, check if **resultShouldBeNegative** is true. If so, return the result of the **applyNegative()** method with **quotient** as a parameter to the calling program.

l. If the program flow reaches the end, then return the current value of **quotient** to the calling program.

Flowchart

The flowcharts for this problem can be seen in *Figures 4.4* and *4.5:*

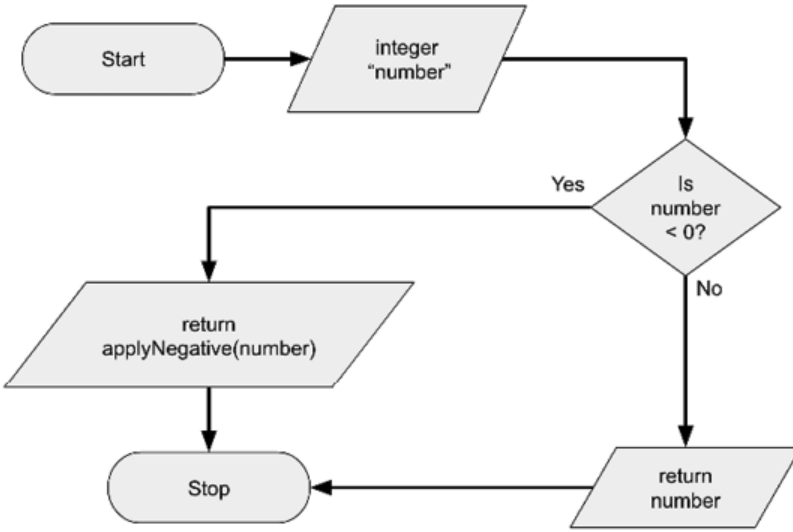

***Figure 4.4**: Flowchart describing how to compute an absolute value*

The following figure shows the step-by-step logic for integer division, determining the result's sign, computing the quotient through repeated subtraction, and returning the final value:

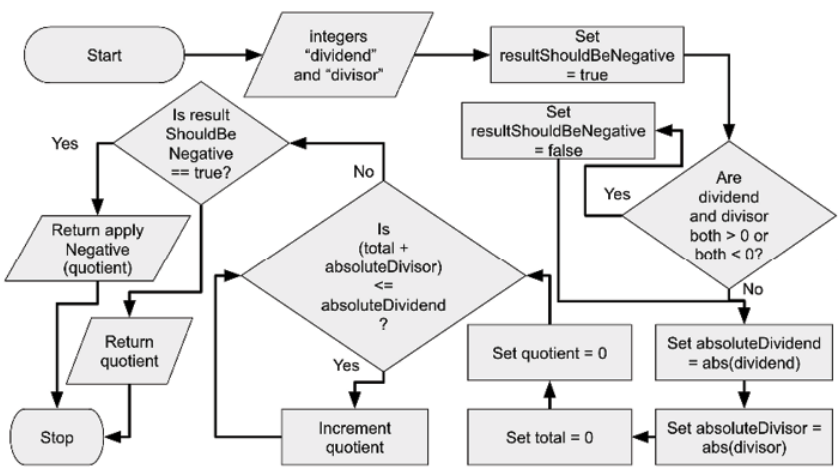

***Figure 4.5**: Flowchart describing how to divide two numbers to get a quotient*

Java code

The Java code for *Program 4.3* is specified as follows:

```
private static int abs(int number) {
```

```
    if (number < 0) {
        return applyNegative(number);
    }
    return number;
}
private static int divide(int dividend, int divisor) {
    boolean resultShouldBeNegative = true;
    // determine ahead of time if the quotient (result) should be
negative
    if ((dividend < 0 && divisor < 0) || (dividend > 0 && divisor >
0)) {
        resultShouldBeNegative = false;
    }
    // for our division calculations we only want to work with
positive numbers
    int absoluteDividend = abs(dividend);
    int absoluteDivisor = abs(divisor);
    int total = 0;
    int quotient = 0;
    while ((total + absoluteDivisor) <= absoluteDividend) {
        total = total + absoluteDivisor;
        quotient++;
    }
    if (resultShouldBeNegative) {
        return applyNegative(quotient);
    }
    return quotient;
}
```

Code explanation

First, we will define a new method named **abs**. This method will

accept an integer parameter named **number**, and return the absolute value of **number** to the calling method. The **abs()** method will prove to be a useful tool in solving our main problem.

Next, we will define a new method named **divide**. This method will accept two integers as parameters, named **dividend** and **divisor**. This method will return the result (quotient) of dividing **dividend** by **divisor**.

Note: This method of division involves only whole numbers, also known as integer division.

Inside the **divide()** method, we will first define a new Boolean variable named **resultShouldBeNegative** with a value of **true**. We will then check if **dividend** and **divisor** are both either greater than zero or both less than zero. If this is the case, our result should be positive, and therefore we will set the value of **resultShouldBeNegative** to false. Essentially, we are taking note of whether the quotient should be positive or negative.

Next, we will define integer variables named **absoluteDividend** and **absoluteDivisor**. They will be set with the result of the **abs()** method using **dividend** and **divisor** as parameters, respectively. The reason that we are doing this is that our division methodology will be easier to understand if we are working with all positive numbers. The integers **absoluteDividend** and **absoluteDivisor** are thus the result of forcing **dividend** and **divisor** to be positive, and the **resultShouldBeNegative** Boolean will help us return the correct sign on the quotient at the end.

We will then build a while loop that iterates as long as the sum of **total** + **absoluteDivisor** is less than or equal to **absoluteDividend**. If it is not, we will add that sum to **total**, and increment **quotient**. The idea here is that our while loop should iterate as long as **absoluteDivisor** keeps fitting into **absoluteDividend**, and the quotient will have our answer when the loop breaks.

Once the loop completes, we will check the value of **resultShouldBeNegative**. It is **true**, then we will return the result of the **applyNegative()** method with **quotient** as a parameter to the calling program. If it is **false**, the program logic flow continues and simply returns the value of the **quotient** to the calling program.

Program 4.4: Check if a number is a power of two

Determine if a given number is a valid power of two. Good examples of powers of two numbers are 2, 4, 8, 16, 32, 64, etc.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new method named **isEven** that accepts an integer named number, and returns a Boolean value. If we use the modulo operator (%) with a divisor of 2, we can check to see if the remainder has a value of 0.
    a. If so, we will return **true** to the calling method.
    b. If the result is not equal to 0, we will return **false**.
2. Write a new method named **isPowerOfTwo** that accepts an integer named **number**, and returns a Boolean value.
3. Define a Boolean variable named **returnVal** with a value of **false**.
4. We will build an if statement with three possible code paths:
    a. First, we will check if the value of the number is equal to 2. If it is, then we will set **returnVal** to true.
    b. Next, we will apply an else/if to check if **number** is not even, by passing it as a parameter to our **isEven()** method. If **number** is not even, we will set **returnVal** to **false**.
    c. Recursively call our **isPowerOfTwo()** method with a value of the number divided by 2 as a parameter. We can use our **divide()** method from *Program 4.3*, passing **number** and 2 as parameters.
5. If the program logic makes it past our multi-tiered if statement, we will simply return the value of **returnVal** to the calling program (or method).

Flowchart

The flowcharts for this problem can be seen in *Figures 4.6 and 4.7:*

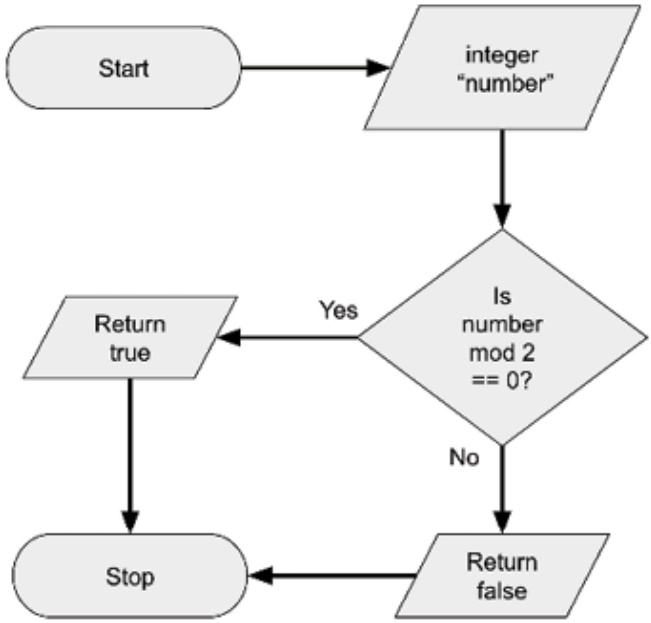

***Figure 4.6**: Flowchart describing how to determine if a number is even*

The following figure shows the logic for checking whether a number is prime, repeatedly dividing it and returning a boolean result based on the checks:

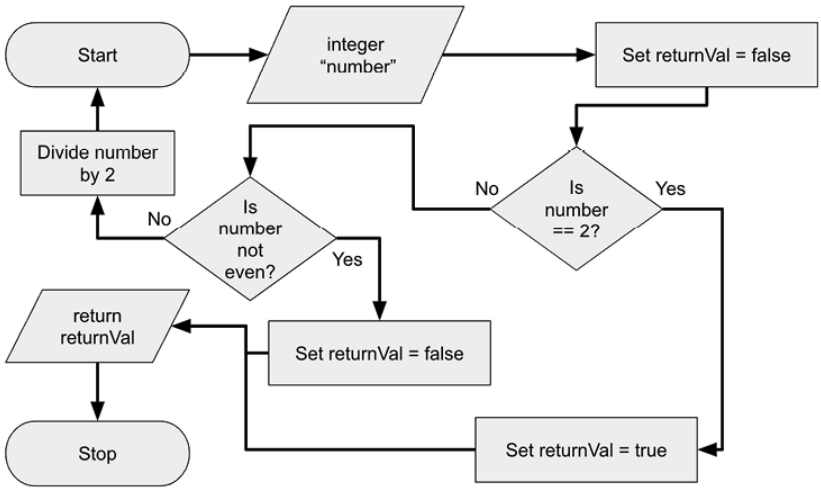

***Figure 4.7**: Flowchart describing how to determine if a number is a power of two*

Java code

The Java code for *Program 4.4* is specified as follows:

```
private static boolean isEven(int number) {
    if (number % 2 == 0) {
        return true;
    }
    return false;
}
private static boolean isPowerOfTwo(int number) {
    boolean returnVal = false;
    if (number == 2) {
        // number is equal to 2
        returnVal = true;
    } else if (number % 2 != 0) {
        // number is odd
        returnVal = false;
    } else {
        // number is not 2 and is not odd,
        // divide number by 2 and check again
        returnVal = isPowerOfTwo(divide(number,2));
    }
    return returnVal;
}
```

Code explanation

First, we will write a new method named **isEven**. It will accept an integer parameter named **number**, and return a Boolean value indicating whether or not the value of **number** is an even number. To accomplish this, we will use the modulo operator (%). Essentially, modulo performs division on a number and returns the remainder. If we divide **number** by 2 and the value of the

remainder is 0, we will return **true**. Otherwise, the number is odd, and we will return **false**.

Next, we will write a new method named **isPowerOfTwo**. It will accept an integer parameter named number, and return a Boolean value indicating whether or not the value of number is a valid power of two number.

Note: Powers of two numbers are one of the fundamental building blocks of computers. Examples throughout the years include the 8-bit Tandy 1000, the 32-bit Sega Saturn, and the 64-bit Intel Core i9.

We will start out by defining a Boolean variable named **returnVal** with a value of **false**. Next, we will check if the **number** is equal to 2. If it is, that means we were successful, and we will set **returnVal** to a value of **true**. Otherwise, we will check if the **number** is odd. We can do this by checking if the result of a call to the **isEven()** method with a parameter of **number** is **false**. If that is the case, we will set **returnVal** to a value of **false**.

If our program flow makes it through those two checks, then we need to further break down the **number** to evaluate it. We will use our **divide()** method with parameters of **number** and 2, and then recursively pass the result back to our same **isPowerOfTwo()** method.

Note: While recursion is a powerful tool, improper use of recursive methods can result in an infinite loop.

Once **number** is broken down enough that it has a value of 2 or it is even, the program's flow will finally make it back to the end of the method, where it will return the value of **returnVal** to the calling method or program. At this point in our program's logical flow, it may have to return to multiple iterations of the **isPowerOfTwo()** method, once for each time it was called from within itself.

Program 4.5: Build a binary error handler

Build an error handling method that takes an integer as a parameter and decodes it into one or more error messages.

Algorithm

To solve this problem, we will take this step: Write a new method named **multiply** that accepts two integers named **numX** and **numY**, and returns an integer.

Flowchart

The flowcharts for this problem can be seen in *Figures 4.8, 4.9, and 4.10:*

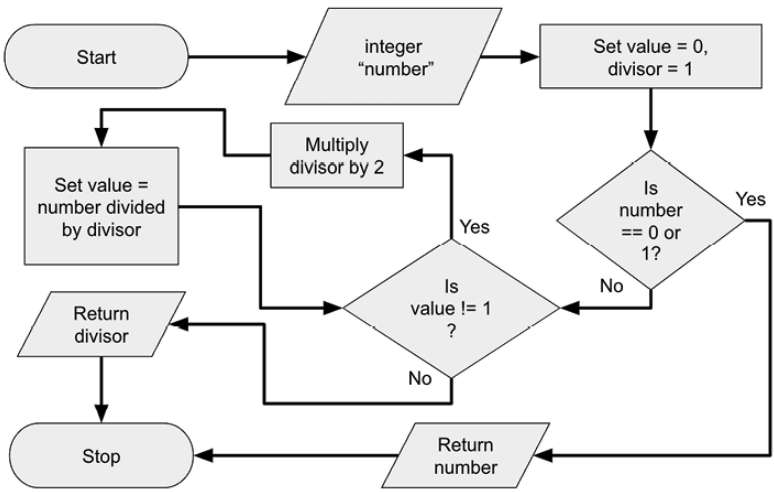

***Figure 4.8**: Flowchart describing how to determine the highest order bit of a number*

The following figure delineates a logical algorithm designed to determine if a specific power of two is a component of a given integer's binary representation:

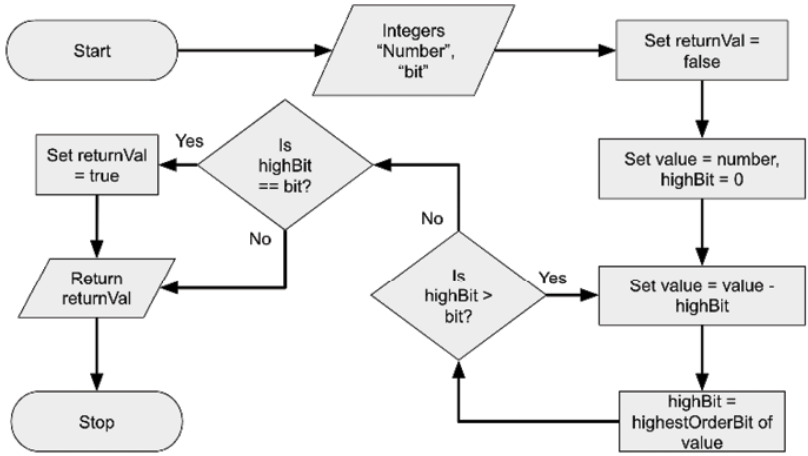

***Figure 4.9**: Flowchart describing how to determine if a specific bit of a number is a 1 or a 0*

This following figure illustrates a diagnostic algorithm that decodes a bitmask integer to identify and return a formatted string of specific error messages:

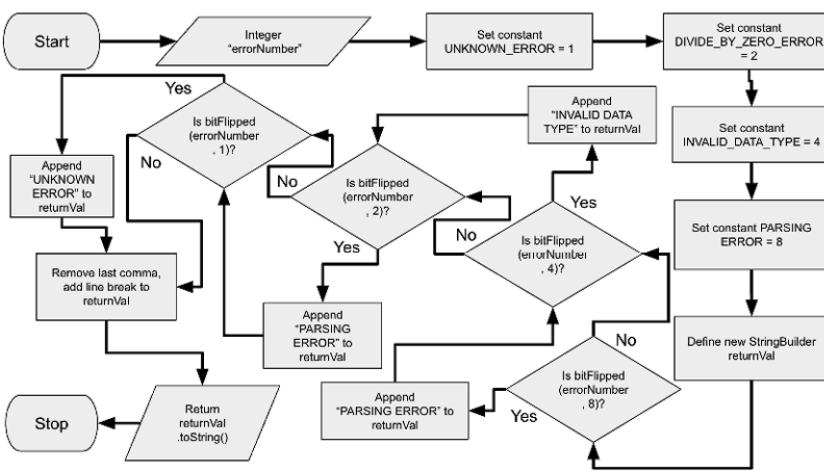

***Figure 4.10**: Flowchart describing how to decipher an error code into its corresponding error messages*

Java code

The Java code for *Program 4.5* is specified as follows:

```
public static int highestOrderBit(int number) {
    int value = 0;
    int divisor = 1;
    if (number == 0 || number == 1) {
        return number;
    }
    while (value != 1) {
        divisor = multiply(divisor, 2);
        value = divide(number, divisor);
    }
    return divisor;
}
public static boolean isBitFlipped(int number, int bit) {
    boolean returnVal = false;
    int value = number;
    int highBit = 0;
    do {
        value -= highBit;
        highBit = highestOrderBit(value);
    } while (highBit > bit);
    if (highBit == bit) {
        returnVal = true;
    }
    return returnVal;
}
private static String getErrorMessages(int errorNumber) {
    final int UNKNOWN_ERROR = 1;
    final int DIVIDE_BY_ZERO_ERROR = 2;
    final int INVALID_DATA_TYPE = 4;
    final int PARSING_ERROR = 8;
```

```
    StringBuilder returnVal = new StringBuilder();
    if (isBitFlipped(errorNumber, PARSING_ERROR)) {
        returnVal.append("PARSING_ERROR, ");
    }
    if (isBitFlipped(errorNumber, INVALID_DATA_TYPE)) {
        returnVal.append("INVALID_DATA_TYPE, ");
    }
    if (isBitFlipped(errorNumber, DIVIDE_BY_ZERO_ERROR)) {
        returnVal.append("DIVIDE_BY_ZERO_ERROR, ");
    }
    if (isBitFlipped(errorNumber, UNKNOWN_ERROR)) {
        returnVal.append("UNKNOWN_ERROR, ");
    }
    if (returnVal.length() > 0) {
        if (returnVal.charAt(returnVal.length() - 2) == ',') {
            return returnVal.substring(0,returnVal.length() - 2) + "\n";
        } else {
            returnVal.append("\n");
        }
    } else {
        returnVal.append("\n");
    }
    return returnVal.toString();
}
```

Code explanation

The general idea behind this problem is that our method will be passed an error code. That error code is then decoded into one or more error messages. The idea is that each binary digit (bit) can represent an individual error message. Therefore, each error message will be assigned to a power of two number. For example, if

our method is passed an error code of 6, then we have to look at the binary representation of 6:

6 = 1 1 0
  4 2 1

In this case, the error code of 6 has values of 1 in its 4th and 2nd bits, indicating that these bits have been flipped. Therefore, we would return error messages for codes 4 and 2.

Let us look at an error code of 8:

8 = 1 0 0 0
  8 4 2 1

In this case, only the 8th bit is *flipped*, so we would only return the error message, the code denoted by the number 8, and no others.

To solve this problem, we will need to build three methods. We will need one method to process the error codes into messages, one method to determine if a specific bit of a number is 1 or 0, and another to determine the highest order bit of a number.

The first method that we will build is to determine the highest-order bit of a number. This method will be named **highestOrderBit**, will accept an integer named **number**, and return an integer value. We will start by defining integer variables named **value** and **divisor**. The **value** variable should have an initial value of 0, and **divisor** should have an initial value of 1.

First, if the **number** is equal to 0 or 1, we will just return **number**. If we are getting a 0, something likely went wrong. But 1 is a legit answer when **number** is equal to 1. We will just make sure to get ahead of that at return 1 right away, without having to spend process time figuring that out. Next, we will build a while loop to run if **value** is not equal to 1. The idea here is that we will keep increasing **divisor** by a factor of 2, while dividing **number** by **divisor** to recompute **value**. If we get a **value** of 1 (regardless of any remainder), that means we have found the highest order bit, and we should return the value of **divisor** to the calling method.

Next, we will need a new method named **isBitFlipped**. This method should accept two integer parameters named **number** and **bit**, and return a Boolean value. The idea behind this method is that we should be able to take a number and a *power of two* bit position and

return true or false whether or not the bit has been *flipped* (has a value of 1, versus 0.

For starters, we will define a Boolean value named **returnVal**. Then we will define an integer named **value** and set it equal to **number**, and also an integer named **highBit** set to 0. We will then build a do/while loop to essentially compute the **highestOrderBit()** of value and keep subtracting **highBit** from **value** until it is no longer greater than bit.

Once we break out of the loop, we will then check whether or not **highBit** is equal to **bit**. If it is, then we know that **bit** has been *flipped* (has a value of 1), and we can set **returnVal** to **true**. Finally, we return the value of **returnVal** back to the calling method.

With those two methods in place, we can finally begin work on our error handling method. We will create a new method named **getErrorMessages**, which accepts an integer parameter named **errorNumber**, and returns a **String**. We will then define a new **StringBuilder** named **returnVal**, as well as the following integers as constants named for each error message, with their error codes as values:

- **UNKNOWN_ERROR = 1**
- **DIVIDE_BY_ZERO_ERROR = 2**
- **INVALID_DATA_TYPE = 4**
- **PARSING_ERROR = 8**

Essentially, we will use the **isBitFlipped()** method to check if the specific power of two bit (specified by each of the error message constants) for **errorNumber** is flipped. If it is, we will append the error message (with a trailing comma and space) to **returnVal**.

Once we have checked for each error message, we will check for and remove the trailing comma from **returnVal**. We will then return the **String** value of **returnVal** using the **toString()** method to the calling program.

Conclusion

In this chapter, we examined problems around basic arithmetic operations and how they work. We built our own methods to perform these operations and even showed how they rely upon each

other. We also briefly looked at recursion and built a recursive method. Additionally, we worked with binary representations of integers and learned how we can use bits of a single number to pass multiple (error) messages at once.

In the next chapter, we will discuss basic data structures. We will have an in-depth look into common structures like stacks, queues, and linked lists. We will also build examples using these structures to solve real-world problems.

Points to remember

Have a look at some key points to remember:

- Checking for input for values that do not need to be computed (like multiplying by 0) and returning early can result in higher performance by short-circuiting (skipping) some of the more processing-intensive areas of a method.
- A negative operation is not intended to force a value to be negative. Applying a negative to a negative will yield a positive.
- There are four parts to a division operation: dividend, divisor, quotient, and remainder.
- Modulo is a division operation that only seeks the remainder. It is also a good way to quickly check if a number is even or odd.
- While recursion is great for solving certain kinds of problems, improper use of recursive methods can result in an infinite loop. Be sure that the method has a way to break out of the recursive loop.

Questions

Answer the following questions to check your understanding:

1. Considering *Program 4.1*, can you think of a way to make it run faster or more efficiently?
2. How would you build an addition method?
3. How would you build a modulo method? In theory, much of

the code from the divide() method could be reused, but what would have to change?

4. Considering the isBitFlipped() method, another way to build it might be with the toBinaryString() static method from the Integer class. What would the code for this new method look like?
5. In the getErrorMessages() method, we check for our error messages with four separate if statements. Could we have used if-else statements instead? Why or why not?
6. In the getErrorMessages() method, we add a comma and a space (for some visual separation) when appending our error messages, leaving us to have to remove the last one. Can you think of another way to return the error messages, separated by commas?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Chapter 5

Basic Data Structures

Introduction

In this chapter, we will examine basic data structures and show how to solve problems using them. We will start with simplistic structures like stacks and queues, and eventually move on to building a linked list.

While it may not always be apparent, these structures are often running in the background of many applications that we use every day. From our CPU's program counter that keeps track of which instructions should be run next, to the back button on a web browser that allows us to navigate back to a page that we were just at, the practical uses are endless.

Basic data structures are literally everywhere in the digital world. Therefore, understanding them and knowing how to use them is a key skill for any software developer.

Structure

This chapter will cover the following programs:

- Program 5.1: Build a stack to process truck pallets

- Program 5.2: Build a queue for airport takeoffs
- Program 5.3: Build a linked list for navigation

Objectives

The main objective of this chapter is to help build a foundational understanding of common data structures. This chapter will also show the reader how to use these structures and apply their principles to solve real problems. For this chapter specifically, we will discuss and demonstrate the usage of stacks, queues, and linked lists.

Note:

1. **The test code given at the beginning of each problem should be added to a new class file, inside of a standard public static void main(String[] args) method. Failure to do so may result in an unnamed class exception. For our purposes, a class named Chapter5 will work just fine.**
2. **It is not uncommon to be asked to build a rudimentary stack or linked list during a technical interview.**

Program 5.1: Build a stack to process truck pallets

Build a stack to accept and manage pallets of products for a retail store being loaded into a truck (lorry). We can test our stack structure with code similar to the following:

```
Stack truckStack = new Stack(8);
truckStack.push("41A - Housewares");
truckStack.push("22F - Personal Care");
truckStack.push("93T - Toys and Games");
truckStack.push("18J - Electronics");
truckStack.push("18K - Electronics");
truckStack.push("81D - Grocery (non-perishable)");
System.out.printf("Next pallet is %s\n", truckStack.peek());
System.out.printf("Unloading %s\n", truckStack.pop());
```

```
System.out.printf("Unloading %s\n", truckStack.pop());
System.out.printf("Unloading %s\n", truckStack.pop());
System.out.printf("Unloading %s\n", truckStack.pop());
System.out.printf("Next pallet is %s\n", truckStack.peek());
System.out.printf("Unloading %s\n", truckStack.pop());
System.out.printf("Unloading %s\n", truckStack.pop());
System.out.printf("Unloading %s\n", truckStack.pop());
```

And we should see output returned which looks like this:

```
Next pallet is 81D - Grocery (non-perishable)
Unloading 81D - Grocery (non-perishable)
Unloading 18K - Electronics
Unloading 18J - Electronics
Unloading 93T - Toys and Games
Next pallet is 22F - Personal Care
Unloading 22F - Personal Care
Unloading 41A - Housewares
Unloading EMPTY
```

Algorithm

To solve this problem, we will take the following steps:

1. Build a new class named **Stack**.
    a. It should have three private variables:
        i. An integer named **maxCount**.
        ii. An integer named **palletCount**.
        iii. A string array named **pallets**.
    b. Build a constructor that accepts a single integer named **maxPallets**. Inside the constructor:
        i. Set the **maxCount** class property to the value of **maxPallets**.
        ii. Initialize the pallets array to be a new string array with the size of **maxPallets**.

c. Create a public **getter** method named **getPalletCount** that returns the value of **palletCount**.

d. Create a public method named **peek** that returns a string. It should:

   i. Check if **palletCount** > **0**. If true, return the string value at **pallets[palletCount – 1]**.

   ii. Otherwise, return the string "**EMPTY**".

e. Create a public method named **pop** that returns a string. It should:

   i. Check if **palletCount** > **0**. If true, decrement **palletCount** and return the string value at **pallets[palletCount – 1]**.

   ii. Otherwise, return the string "**EMPTY**".

f. Create a public method named **push** that accepts a string named *pallet* as its parameter. It should:

   i. Check if **palletCount** < **maxCount**. If true, set **pallets[palletCount]** = **pallet** and increment **palletCount**.

Flowchart

The flowcharts for *Program 5.1* can be seen in the following *Figures 5.1*, *5.2*, and *5.3*:

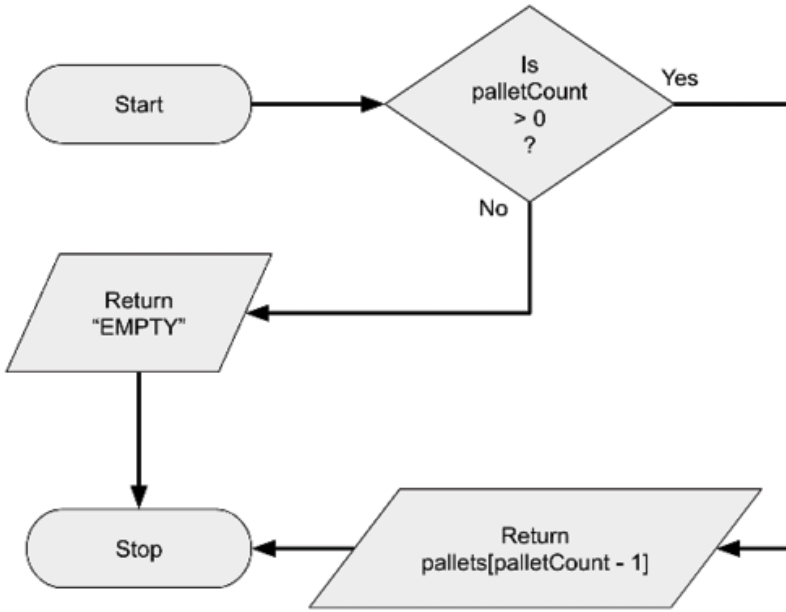

Figure 5.1: *Flowchart describing the peek() method of a stack*

The following figure shows the decision flow for retrieving a pallet, checking availability first and handling both empty and in-stock cases clearly:

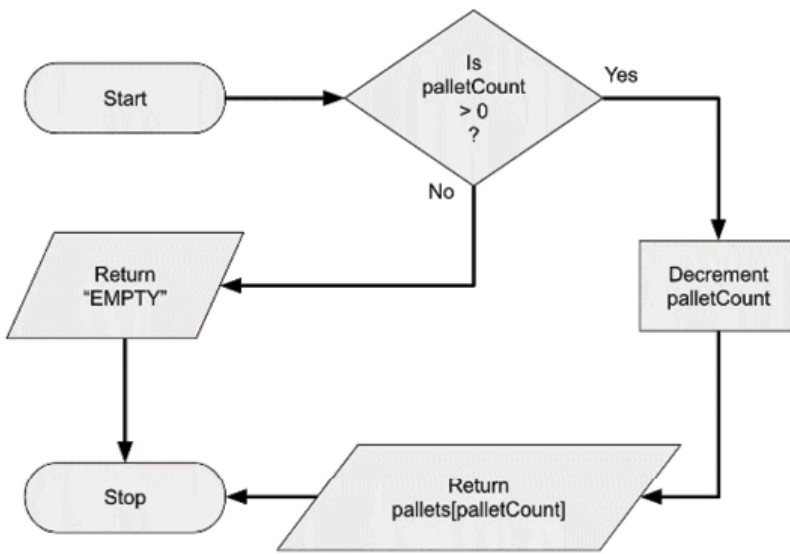

Figure 5.2: *Flowchart describing the pop() method of a stack*

The following figure shows the decision flow for adding a pallet, ensuring capacity limits are respected before updating the count and storing the item:

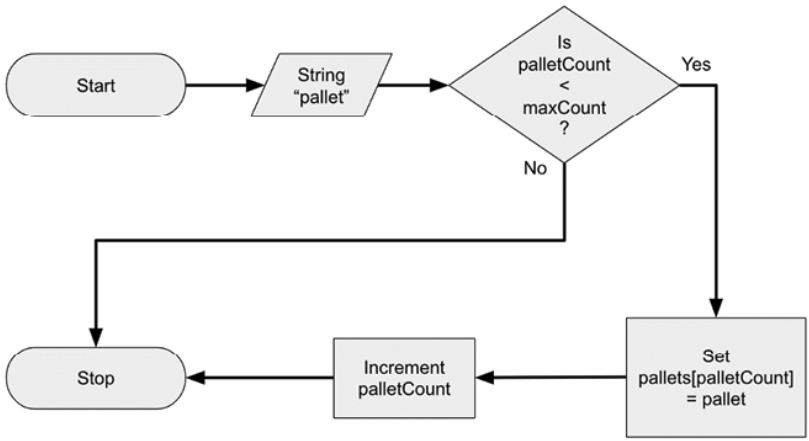

Figure 5.3: *Flowchart describing the push() method of a stack*

Java code

The Java code for *Program 5.1*'s stack class is specified in the following section:

```
public class Stack {
    private int maxCount = 0;
    private int palletCount = 0;
    private String[] pallets;
    public Stack(int maxPallets) {
        this.maxCount = maxPallets;
        pallets = new String[maxPallets];
    }
    public int getPalletCount() {
        return palletCount;
    }
```

```
    public String peek() {
        if (palletCount > 0) {
            return pallets[palletCount - 1];
        } else {
            return "EMPTY";
        }
    }
    public String pop() {
        if (palletCount > 0) {
            palletCount--;
            return pallets[palletCount];
        } else {
            return "EMPTY";
        }
    }
    public void push(String pallet) {
        if (palletCount < maxCount) {
            pallets[palletCount] = pallet;
            palletCount++;
        }
    }
}
```

Code explanation

For this problem, we will need to build a stack class. A stack's fundamental premise is that it processes its contents on a **last-in-first-out** (**LIFO**) basis. The first item pushed onto the stack is on the bottom, and all other items line up on top of it. Items are removed or popped from the top of the stack, with the first item being the last to be removed.

In our case, we will apply the stack paradigm to loading pallets into a truck. After all, it is often impossible to unload the first pallet

placed without unloading the pallets in front of it.

The **Stack** class will need three private properties: integers named **maxCount** and **palletCount** (both initialized to 0), and a string array named **pallets**. We will also need a constructor that accepts an integer named **maxPallets** as a parameter. This constructor will set **maxCount** to the value of **maxPallets**, and then initialize **pallets** to be a new string array with a size of **maxPallets**.

Note: We could use either maxCount or maxPallets to set the size of the pallets array.

Next, we will build a public **getter** accessor for the **maxCount** property. This will allow users of this class to get a count of how many **pallets** of items are currently in the truck.

Moving on, we will build a new method named **peek** that does not accept any parameters and returns a string. When we want to see the top or next item in the stack (without adding or removing anything), the **peek()** method lets us do that. We will first check to ensure that **palletCount** is $>$ 0. If so, we will return the string referenced by **pallets[palletCount – 1]**. Essentially, if there is 1 item in the **pallets** array, we are returning the string at **pallets[0]**. If **palletCount** is not $>$ 0 (the truck is empty), we will simply return the string **"EMPTY"**.

To remove an item from the stack, we will **pop** it. Write a new method named **pop** that does not accept any parameters and returns a string. The **pop()** method will actually be very similar to the **peek()** method, except that **pop()** will reduce the number of items in the stack. Similarly, we will first check that **palletCount** is $>$ 0. If it is, we will decrement **palletCount**, and then return the string at **pallets[palletCount]**. Otherwise, if **palletCount** is not $>$ 0 (the truck is empty), we will return the string **"EMPTY"**.

Note: The pop() method does not actually remove the target string from the pallets array; it simply decrements palletCount so that it cannot be accessed. This is called a soft delete.

Finally, we need a method that lets us push items onto the stack. We will write a new method named **push**, which will accept a string named **pallet**. It will first make sure that **palletCount** is $<$ **maxCount**. After all, our truck has a maximum number of pallets

that it can safely carry. If true, we will set **pallets[palletCount] = pallet**, and then increment **palletCount**.

Program 5.2: Build a queue for airport takeoffs

Build a queue to accept and manage the order of aircraft takeoffs for an airport. We should be able to test it with a code similar to the following:

```
Queue takeoff = new Queue(10);
System.out.printf("Routing %s to takeoff runway.\n",
takeoff.enqueue("DL2150"));
System.out.printf("Routing %s to takeoff runway.\n",
takeoff.enqueue("AA1822"));
System.out.printf("Now taking off is %s, and %s is next.\n",
takeoff.dequeue(), takeoff.getFront());
System.out.printf("Routing %s to takeoff runway.\n",
takeoff.enqueue("DL2205"));
System.out.printf("Routing %s to takeoff runway.\n",
takeoff.enqueue("DL1141"));
System.out.printf("Now taking off is %s, and %s is next.\n",
takeoff.dequeue(), takeoff.getFront());
System.out.printf("Routing %s to takeoff runway.\n",
takeoff.enqueue("SW1004"));
System.out.printf("Now taking off is %s, and %s is next.\n",
takeoff.dequeue(), takeoff.getFront());
```

When run, this code should produce results similar to the following output:

```
Routing DL2150 to takeoff runway.
Routing AA1822 to takeoff runway.
Now taking off is DL2150, and AA1822 is next.
Routing DL2205 to takeoff runway.
Routing DL1141 to takeoff runway.
Now taking off is AA1822, and DL2205 is next.
```

Routing SW1004 to takeoff runway.

Now taking off is DL2205, and DL1141 is next.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new class named **Queue**. This class will have two private variables:
    a. A string array named **aircraft**.
    b. An integer named **currentCount**.
2. Write a new constructor method for the **Queue** class that accepts an integer named **maxCount** as a parameter. Inside the constructor:
    a. Instantiate **aircraft** with a new string array with a size of **maxCount**.
3. Write a new public method named **enqueue** that accepts a string named **name** as a parameter, and returns a string value. Inside this method, do the following:
    a. Set the string at position **currentCount** in **aircraft** to the value of **name**.
    b. Increment **currentCount**.
4. Write a new public method named **getFront** that does not accept any parameters and returns a string value. Inside this method:
    a. If **currentCount** is > 0, return the string value at **aircraft[0]** to the calling method.
    b. Otherwise, return the message **Queue is empty** to the calling method.
5. Write a new public method named **dequeue** that does not accept any parameters and returns a string value. Inside this method, do the following:
    a. If **currentCount** is > 0:
        i. Define a new string named **front** with the value returned by the **getFront()** method.

b. Otherwise, return the message **Queue is empty** to the calling method.

Flowchart

The flowcharts for this problem can be seen in the following *Figures 5.4*, *5.5*, and *5.6*:

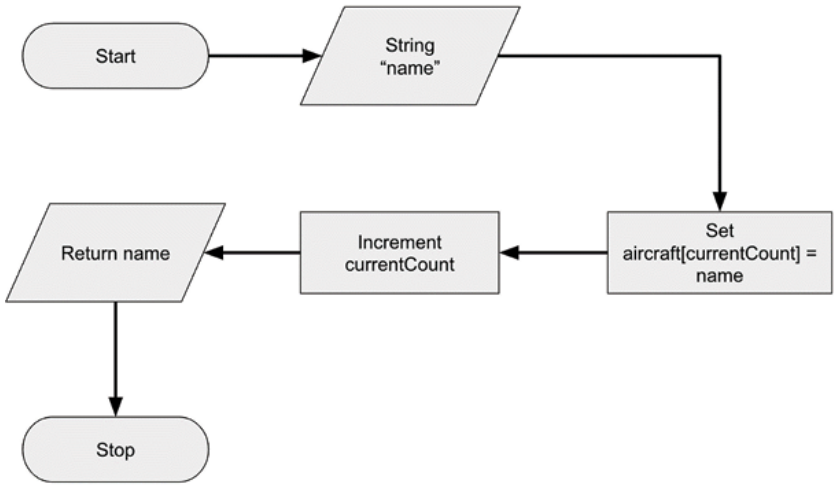

Figure 5.4: *Flowchart describing the enqueue() method*

The following figure shows the decision flow for reading from the queue, returning the next aircraft when available or indicating that the queue is empty:

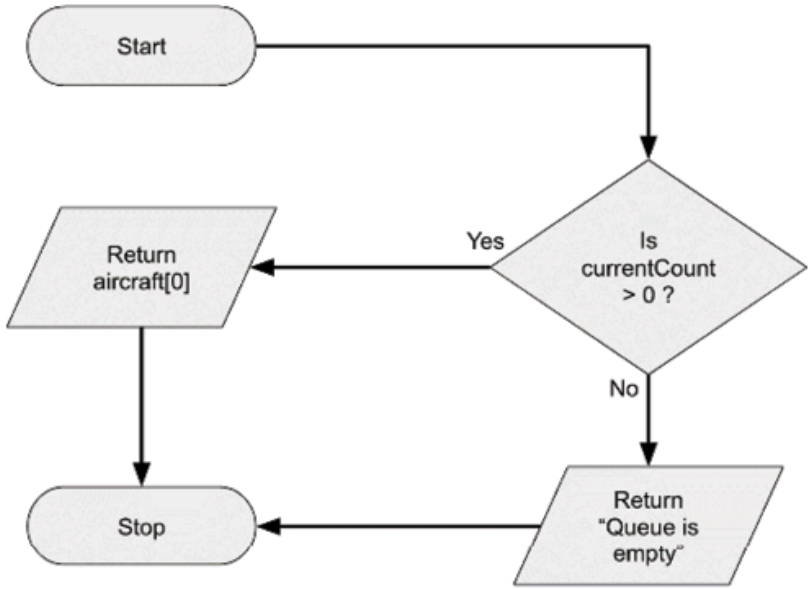

Figure 5.5: *Flowchart describing the getFront() method*

The following figure shows the decision flow for removing an element from the queue, shifting remaining items forward and handling the empty-queue case safely:

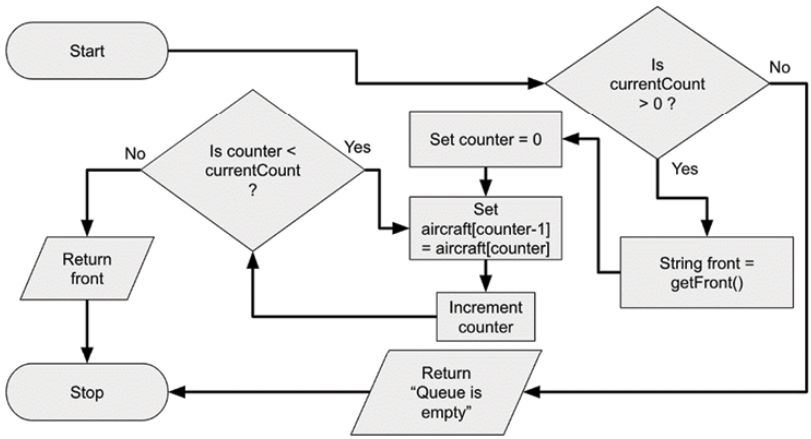

Figure 5.6: *Flowchart describing the* dequeue() *method*

Java code

The Java code for *Program 5.2*'s queue class is specified in the following section:

```
public class Queue {
    private String[] aircraft;
    private int currentCount = 0;
    public Queue(int maxCount) {
        aircraft = new String[maxCount];
    }
    public String enqueue(String name) {
        aircraft[currentCount] = name;
        currentCount++;
        return name;
    }
    public String getFront() {
        if (currentCount > 0) {
            return aircraft[0];
        } else {
            return "Queue is empty";
        }
    }
    public String dequeue() {
        if (currentCount > 0) {
            String front = getFront();
            for (int counter = 1; counter < currentCount; counter++) {
                aircraft[counter - 1] = aircraft[counter];
            }
            currentCount--;
            return front;
        } else {
```

```
            return "Queue is empty";
        }
    }
}
```

Code explanation

To solve this problem, we are going to build a separate class, named **Queue**. A queue differs from a stack, as it processes its items on a **first-in-first-out** (**FIFO**) basis. That is, the first item in the queue is the first one out, and all new items line up behind it in order.

Our **Queue** class will need two private variables. One is a string array named **aircraft**, the other is an integer named **currentCount** (initialized to 0). The constructor for the **Queue** class will accept an integer named **maxCount** as a parameter. The constructor's sole job is to instantiate **aircraft** into a new string array with a size of **maxCount**.

Note: It is not a bad idea to have an additional constructor for the Queue class that does not have any parameters and initializes the aircraft array to a default size.

Next, we will build a new method named **enqueue** that accepts a string parameter named **name**, and returns a string value. This method takes the name/identifier of a particular **aircraft** and adds it to the back of the queue. Simply put, it will set **aircraft[currentCount]** to the value of the **name** parameter. With that complete, the method then increments the value of **currentCount** by 1, and returns **name** to the calling method.

Note: Remember, arrays and collections in Java are zero-based. For us, that means currentCount contains both the current number of aircraft in the queue, as well as the index value of the next aircraft.

We will then build a new method named **getFront** that does not have any parameters and returns a string value. This method will return the name of the aircraft that is at the front/beginning of the **aircraft** array (at position 0). The **getFront()** method will first make sure that the **aircraft** array is not empty (**currentCount** > **0**), and then return the string value at **aircraft[0]** to the calling method. If the **aircraft** array is empty, we will return a message of **Queue is**

empty to the calling method.

Finally, we will build a method named **dequeue** that does not have any parameters and returns a string value. This method will remove an aircraft from the queue and return it. As was done with the **getFront()** method, we will first make sure that the **aircraft** array is not empty. If it is empty, we will return a message of **Queue is emtpy** to the calling method.

If it is not empty, we will instantiate a new string named **front** with the value returned from the **getFront()** method. We will then iterate through all items in the **aircraft** array, except for the first one. Therefore, our for loop will iterate a **counter** from 1 as long as the counter is less than **currentCount**. Inside the loop, we will set **aircraft[counter – 1]** to the value of **aircraft[counter]**, essentially moving each string in the array one index position closer to the front. With the reshuffling of array contents complete, we will decrement **currentCounter** by 1, and return the value of **front** to the calling method.

Program 5.3: Build a linked list for navigation

Build a linked list structure to manage step-by-step automotive driving directions. We will need Java classes to manage the linked list itself, as well as the individual driving steps. To test it, we could implement a code similar to the following:

```
LinkedList directions = new LinkedList();
Step step1 = new Step(UUID.randomUUID(),
   "After 0.1 miles, turn left onto Wuthering Hills Drive.");
Step step2 = new Step(UUID.randomUUID(),
   "After 1.3 miles, turn right onto Highway 11.");
Step step3 = new Step(UUID.randomUUID(),
   "After 0.3 miles, exit the roundabout onto the ramp for I-39S/
I-90E.");
Step step4 = new Step(UUID.randomUUID(),
   "After 12 miles, take exit 1 for IL-75W/US-51N, and turn right.");
Step step5 = new Step(UUID.randomUUID(),
```

```
    "After 1.2 miles, turn right onto US-51N. You are now entering
Wisconsin.");
Step step6 = new Step(UUID.randomUUID(),
    "After 1.1 miles, turn right onto Pleasant St.");
Step step7 = new Step(UUID.randomUUID(),
    "After 350 feet, turn left onto East Grand Ave.");
Step step8 = new Step(UUID.randomUUID(),
    "After 0.2 miles, turn right onto 2nd St.");
Step step9 = new Step(UUID.randomUUID(),
    "After 350 feet, you have arrived at 705 3rd St. Beloit, WI.");
directions.addStep(step1);
directions.addStep(step2);
directions.addStep(step3);
directions.addStep(step4);
directions.addStep(step5);
directions.addStep(step6);
directions.addStep(step7);
directions.addStep(step8);
directions.addStep(step9);
printDirections(directions);
```

And we would expect results similar to the following:

After 0.1 miles, turn left onto Wuthering Hills Drive.

After 1.3 miles, turn right onto Highway 11.

After 0.3 miles, exit the roundabout onto the ramp for I-39S/ I-90E.

After 12 miles, take exit 1 for IL-75W/US-51N, and turn right.

After 1.2 miles, turn right onto US-51N. You are now entering Wisconsin.

After 1.1 miles, turn right onto Pleasant St.

After 350 feet, turn left onto East Grand Ave.

After 0.2 miles, turn right onto 2nd St.

After 350 feet, you have arrived at 705 3rd St. Beloit, WI.

Algorithm

To solve this problem, we will take the following steps:

1. Write a new class named **Step**:
    a. Inside the class, define three private properties:
        i. A UUID (**import java.util.UUID**) named **id**.
        ii. A string named **description**.
        iii. A self-referential property of the **Step** class named **nextStep**.
    b. Define public getters and setters for those properties.
    c. Write a constructor for the **Step** class that accepts parameters of **id** (UUID) and **desc** (string).
        i. The constructor should use these parameters (**id** and **desc**) to set the class properties of **id** and **description**, respectively.
2. Write a new class named **LinkedList**:
    a. If **stepCount** is not > 0, set **firstStep** and **lastStep** both equal to **step**.
    b. Increment **stepCount**.

Flowchart

The flowcharts for this problem can be seen in the following *Figures 5.7* and *5.8:*

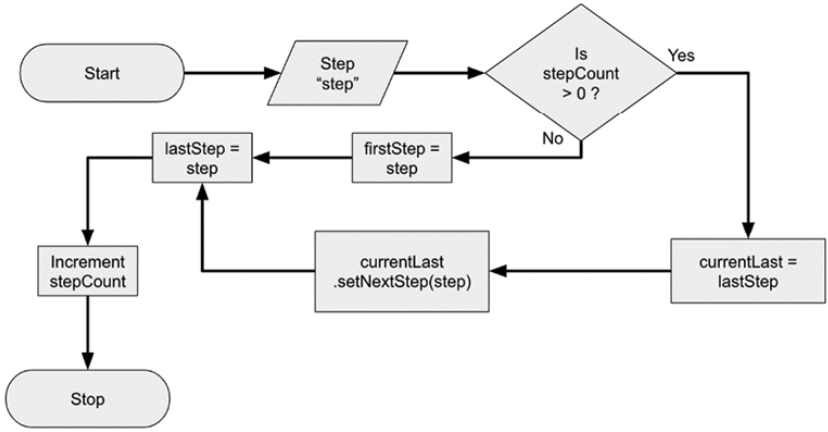

Figure 5.7: *Flowchart describing how to add steps to the (linked) list of directions*

The following figure shows the step-by-step traversal of a linked list, printing each direction in sequence until no further steps remain.

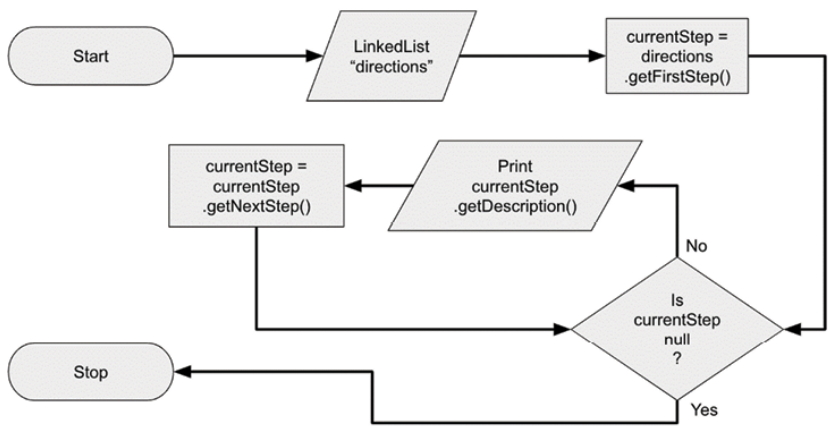

Figure 5.8*: Flowchart describing how to traverse the steps in the (linked) list with the printDirections() method*

Java code

The Java code for *Program 5.3* is specified in the following section:

```
import java.util.UUID;
public class Step {
```

```
    private UUID id;
    private String description;
    private Step nextStep;
    public Step(UUID id, String desc) {
        this.id = id;
        this.description = desc;
    }
    public UUID getId() {
        return id;
    }
    public void setId(UUID id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Step getNextStep() {
        return nextStep;
    }
    public void setNextStep(Step nextStep) {
        this.nextStep = nextStep;
    }
}
public class LinkedList {
    private Step firstStep;
    private Step lastStep;
    private int stepCount = 0;
```

```
    public void addStep(Step step) {
        if (stepCount > 0) {
            Step currentLast = lastStep;
            currentLast.setNextStep(step);
            lastStep = step;
        } else {
            setFirstStep(step);
        }
        stepCount++;
    }
    public Step getFirstStep() {
        return firstStep;
    }
    public Step getLastStep() {
        return lastStep;
    }
    private void setFirstStep(Step step) {
        firstStep = step;
        lastStep = step;
    }
}
private static void printDirections(LinkedList directions) {
    Step currentStep = directions.getFirstStep();
    while (currentStep != null) {
        System.out.printf("%s\n", currentStep.getDescription());
        currentStep = currentStep.getNextStep();
    }
}
```

Code explanation

This program will require two object classes, named **Step** and **LinkedList**. We will start with the **Step** class, which will have three private properties: A **Universal Unique Identifier** (**UUID**) named **id**, a string named **description**, and a self-referential **Step** object named *step*.

Note: There are several types of UUIDs. We will be working with a type-4 UUID, which is a 128-bit number with a random component. We will need to import the java.util.UUID library to use it.

To make things easy, the **Step** class will have only one constructor, which accepts two parameters: a UUID named **id** and a **String** named **desc**. The constructor method will set the class **id** property to the value of the **id** parameter. It will also set the class **description** property to the value of the **desc** parameter.

Note: Class properties and method parameters can have the same names, as long as the class property is prefaced with the this keyword, which is a way to directly reference the current instance of the class.

The remaining code in the **Step** class is for the accessor methods (getters and setters) of the properties. Examples are **getId()**, **setId()**, **getDescription()**, **setDescription**, **getNextStep()**, and **setNextStep()**. The getters essentially only return the class property that they are named for. The setters accept a lone parameter of the same type (and often name) as the corresponding class properties and set the property to the parameter value. These will be used later.

Note: Each step in our linked list has a self-referential object defined for the next step. This means that we have a one-way linked list, which can only be traversed in one direction. For a two-way linked list, we would also have to keep a self-referential object defined for the previous step as well.

Next, we will build the **LinkedList** class. This class will have three private properties: an object of the **Step** class named **firstStep**, an object of the **Step** class named **lastStep**, and an integer named **stepCount** set to 0.

The **LinkedList** class does not need a constructor defined, so we will not include one. This class also will not need any setter accessors, but we will need two getters. Therefore, we will define a **getFirstStep** method and a **getLastStep** method, which simply return

firstStep and **lastStep**, respectively.

Next, we will create a method named **addStep** that accepts a **Step** object named **step**. Inside this method, we will first check to see if **stepCount** > **0**. If so, we will define a new **Step** object named **currentLast**, and set it to **lastStep**. We will then call the **setNextStep()** method on **currentLast** object to set **step** as the last step in the list.

If **stepCount** is not greater than 0, then the list is empty. In this case, we will simply set **firstStep** equal to **step**. Once the if construct is complete, we will set **lastStep** to **step** and then increment **stepCount**. In this way, our **addStep()** method always adds new steps to the back of the list.

To traverse through our linked list, we will write a method named **printDirections** in our main program, which accepts a **LinkedList** object named **directions** as a parameter. It will first define a **Step** object named **currentStep**, and instantiate it with the result of the **getFirstStep()** method of the **directions** object.

Next, we will build a while loop that runs as long as **currentStep** is not equal to null. Inside the loop, we will print the description of **currentStep**, using its **getDescription()** method. The only other instruction to run before the loop restarts is to set **currentStep** to the result of the **getNextStep()** method of **currentStep**.

In our linked list, the order is kept because each item (step) only knows about the step behind it. As long as we have a reference to the front, we use each step's **getNextStep()** method to iterate through the list. In this way, we traverse the list moving from step to step.

Conclusion

In this chapter, we introduced basic data structures, specifically stacks, queues, and linked lists. We also discussed real-world scenarios that can be solved with the mentioned data structures. Additionally, we demonstrated how to leverage the fundamental concepts of each of those structures and even built (minimal) example solutions.

These basic data structures are everywhere in the paradigm of

computing. Once we have an understanding of them, it is not uncommon to start recognizing their ubiquity and the roles they play in systems that we use every day.

In the next chapter, we will change our focus to pattern matching. We will discuss regular expressions, including their capabilities and nuances. We will also look at how to match on different object/instance types with the switch/case construct, which is a relatively new feature in Java.

Points to remember

Have a look at some key points to remember:

- The fundamental premise of FIFO structures (like queues) is that the first items added to the structures are also the first items to be processed out of the structures. Essentially, items are processed in the order that they are received.
- LIFO structures (like stacks) will process the most recently added items first, leaving the very first item added for last.
- If a Java class does not have a defined constructor, a no-parameter constructor (that performs a rudimentary instantiation of the class) is assumed.
- Common operations on a stack are peek, pop, and push.
- UUIDs are a great way to ensure that entity identifiers are both random and unique.
- The core principle of a linked list is that each item only knows about its adjacent items.
- Items in a one-way linked list are linked to the next item and can only be traversed in a single direction. Whereas items in a two-way linked list are linked to both the next and previous items, and can be traversed either front-to-back or back-to-front.

Questions

Answer the following questions to check your understanding:

1. Considering Program 5.1, what would happen when we add more pallets than maxCount? Can you write a method that would resize the underlying array?
2. The Stack class as shown does not have a setter for palletCount. Should it have one? Why or why not?
3. Considering Program 5.2, how would you implement a getBack() method to view the most-recently added aircraft in the queue?
4. Can you write an additional constructor for the Queue class that does not accept any parameters and defaults the underlying string array to 10 aircraft?
5. Considering Program 5.3, how could we remove a step from the linked list? Can you write a method to do this?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Chapter 6

Pattern Matching

Introduction

A common developer's task is to take the data from one source and import it into another data store. The data in these scenarios is almost never an exact fit with the new system and requires some work before an import is considered successful. A developer charged with fixing seven or eight text issues in a file could do that manually. However, for even seven or eight hundred such issues, a manual approach is simply not feasible.

The focus of this chapter will be on different ways to leverage various pattern-matching tools to solve problems. While the tools at a developer's disposal will vary by language, there are some commonalities. For instance, we will spend some time in this chapter looking at regular expressions, which are fairly ubiquitous across most programming languages. We will also examine some Java-specific approaches, such as pattern matching with the switch/case construct.

The actual Java code for all of the regular expression-based solutions is going to be extremely similar, if not exact. The flowcharts for these problems will reflect this as well. However, the

real learning from these problems is going to come from reading the code explanations, where each of the regular expression patterns will be broken down and explained.

Structure

This chapter will cover the following programs:

- Program 6.1: Determine a valid phone number
- Program 6.2: Determine if a new password is valid
- Program 6.3: Determine if an IP address is valid
- Program 6.4: Compute tollway charges by vehicle

Objectives

The primary goal of this chapter is to introduce pattern matching into the developer's personal toolkit. The programs within have the aim of imparting an understanding of the concepts for pattern matching, so that the reader can use them to solve real-world computing problems. To that end, we will work with some regular expression patterns of increasing complexity and have a look at matching instance types with a switch/case statement.

Program 6.1: Determine a valid phone number

Using regular expressions, build a method to check whether user-entered phone numbers are valid. We can test it with code similar to the following:

```
String phoneNumber = "4144524824";
String phoneNumber2 = "(313)472-3929";
String phoneNumber3 = "313-459-7199";
String phoneNumber4 = "331234567";
String phoneNumber5 = "414.458.1556";
String phoneNumber6 = "44-458-3831";
String phoneNumber7 = "(414)882-113";
```

```
Pattern phonePattern =
        Pattern.compile("[(.]*[0-9]{3}[)-.]*[0-9]{3}[-.]*[0-9]{4}");
System.out.printf("is %s valid? %b\n", phoneNumber,
        isValidPhoneNumber(phoneNumber, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber2,
        isValidPhoneNumber(phoneNumber2, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber3,
        isValidPhoneNumber(phoneNumber3, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber4,
        isValidPhoneNumber(phoneNumber4, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber5,
        isValidPhoneNumber(phoneNumber5, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber6,
        isValidPhoneNumber(phoneNumber6, phonePattern));
System.out.printf("is %s valid? %b\n", phoneNumber7,
        isValidPhoneNumber(phoneNumber7, phonePattern));
```

This should produce an output similar to the following:

```
is 4144524824 valid? true
is (313)472-3929 valid? true
is 313-459-7199 valid? true
is 331234567 valid? false
is 414.458.1556 valid? true
is 44-458-3831 valid? false
is (414)882-113 valid? False
```

Algorithm

To solve this problem, we will take the following steps:

1. Define a new **Pattern** object named **phonePattern** instantiated with the result of **Pattern**'s **compile()** static method.
    a. The **compile()** method should have the following regular

expression passed as a (**String**) parameter: **[(.]*[0-9]{3}[)-.]*[0-9]{3}[-.]*[0-9]{4}**

2. Build a new method named **isValidPhoneNumber**, that accepts a **String** named **number** and a **Pattern** object named **pattern**. Inside this method, do the following:
   a. Define a new **Matcher** object named **phoneMatcher** instantiated with the result **pattern**'s **matcher()** method, while passing **number** as a parameter.
   b. Return the result of **phoneMatcher.matches()** to the calling program.

Flowchart

The flowchart for *Program 6.1* can be seen in *Figure 6.1:*

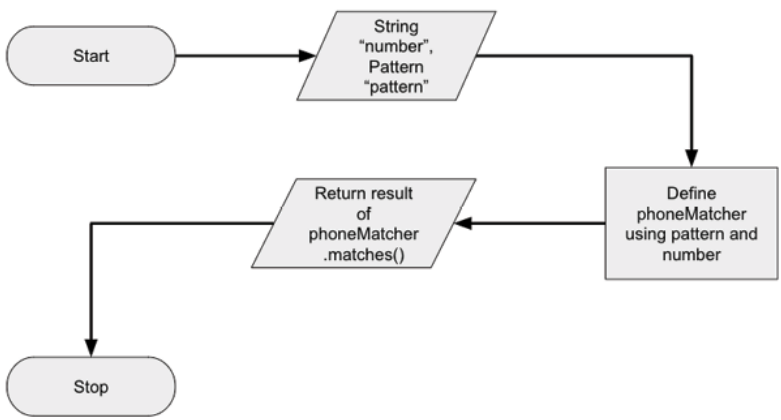

***Figure 6.1**: Flowchart describing the isValidPhoneNumber() method*

Java code

The Java code for *Program 6.1's* **isValidPhoneNumber()** method is specified in the following code:

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
Pattern phonePattern =
```

```
        Pattern.compile("[(.]*[0-9]{3}[)-.]*[0-9]{3}[-.]*[0-9]{4}");
    private static boolean isValidPhoneNumber(String number, Pattern
    pattern) {
        Matcher phoneMatcher = pattern.matcher(number);
        return phoneMatcher.find();
    }
```

Note: The previous code is meant to be in the same class file as the test code shown at the beginning of this program. While the isValidPhoneNumber() method is fine on its own, the phonePattern should be defined in the class main() method, before calls to the isValidPhoneNumber() method. Please see the code addendum for this chapter for more details.

Code explanation

For this problem, we will need to instantiate a new **Pattern** object named **phonePattern**. It should be instantiated with the result of **Pattern**'s **compile()** static method, with the following regular expression passed as a parameter:

[(.]*[0-9]{3}[)-.]*[0-9]{3}[-.]*[0-9]{4}

Note: As mentioned in Chapter 3, Characters and Strings, the pattern.compile() operation consumes a not-so-insignificant amount of resources. Therefore, it is better to call it outside of our method and pass it as a parameter so that we are only calling it once for each pattern. Then we can use it as many times as we need to.

The regular expression in the preceding section works in the following manner:

- **[(]***: A phone number may start with the area code encapsulated in parentheses. Therefore, we will begin by checking for a left parenthesis. We can do that by specifying the left parenthesis inside of a pair of brackets (used to define a set of one or more characters to match). We will also add an asterisk, which indicates that we want to match zero or more characters of the preceding character set. After all, if the phone number that we are validating was not defined with the area code in parentheses, we do not want it to be immediately disqualified.

- **[0-9]{3}**: This section validates the 3-digit area code. We want to match on digits 0 through 9, so those are encapsulated in brackets. We also want to match on exactly 3 of them, so we have added a 3 encapsulated in braces.
- **[)-.]*:** Next, we will check for the presence of a separator between the area code and the prefix code. Valid characters will either be a right parenthesis, a dash, or a dot (period). We will define that character set by encapsulating those characters inside of brackets. Of course, we could also get a phone number typed in without any separators, so we will add an asterisk to indicate that the separator may or may not be there.
- **[0-9]{3}**: Similar to how we checked for the area code, we will also check for the 3-digit prefix code. Again, we will define the character set to check for the digits 0 through 9, and we will add a 3 encapsulated in braces because we want exactly 3 of them.
- **[-.]*:** The check for the separator between the prefix code and the line code is a little different, because we no longer have to check for parentheses. Here, we will define a character set of only a dash and a dot, followed by an asterisk (as a separator does not have to be present).
- **[0-9]{4}**: Finally, we need to match on the last 4 digits (also known as the *line code*). We will define the character set to check for digits 0 through 9, and make sure that we get 4 of them.

With our regular expression pattern defined, we will write a new method named **isValidPhoneNumber**. This method should return a Boolean value, and it should accept a string named **number** and a **Pattern** named **pattern** as its parameters.

Inside the **isValidPhoneNumber()** method, we will define a new **Matcher** object named **phoneMatcher**, instantiated with the result from the **pattern**'s **matcher()** method, using number as a parameter. Finally, we will return the result of **phoneMatcher**'s **matches()** method to the calling program.

Note: The Matcher class has other, similar methods available for

determining a match on a string with a regular expression. We are using the matches() method because it matches the pattern against the entire string, not just a subset of it.

Program 6.2: Determine if a new password is valid

Using regular expressions, build a method to determine if a newly created password meets the following requirements:

- At least 1 lowercase character.
- At least 1 uppercase character.
- At least 1 number.
- Between 12 and 25 characters in length.

We can test our method with code similar to the following:

```
String password1 = "blahblah";
String password2 = "blahBlah@1218";
String password3 = "correctHorseBattery";
String password4 = "corr3ctHors3Batt3ry";
Pattern passwordPattern = Pattern
    .compile("(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).{12,25}");
System.out.printf("is %s a valid password? %b\n", password1,
    isValidPassword(password1, passwordPattern));
System.out.printf("is %s a valid password? %b\n", password2,
    isValidPassword(password2, passwordPattern));
System.out.printf("is %s a valid password? %b\n", password3,
    isValidPassword(password3, passwordPattern));
System.out.printf("is %s a valid password? %b\n", password4,
    isValidPassword(password4, passwordPattern));
```

This should produce an output similar to the following:

```
is blahblah a valid password? False
is blahBlah@1218 a valid password? true
```

```
is correctHorseBattery a valid password? false
is corr3ctHors3Batt3ry a valid password? True
```

Algorithm

To solve this problem, we will take the following steps:

1. Define a new **Pattern** object named **passwordPattern**, instantiated with the result of **Pattern**'s **compile()** static method.
    a. The **compile()** method should have the following regular expression passed as a parameter: **(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).{12,25}**
2. Build a new method named **isValidPassword** that accepts a string named **password** and a **Pattern** named **pattern**. Inside this method, do the following:
    a. Define a new **Matcher** object named **passwordMatcher** instantiated with the result of **pattern**'s **matcher()** method, while passing **password** as a parameter.
    b. Return the result of **passwordMatcher.matches()** to the calling program.

Flowchart

The flowchart for *Program 6.2* can be seen in *Figure 6.2*:

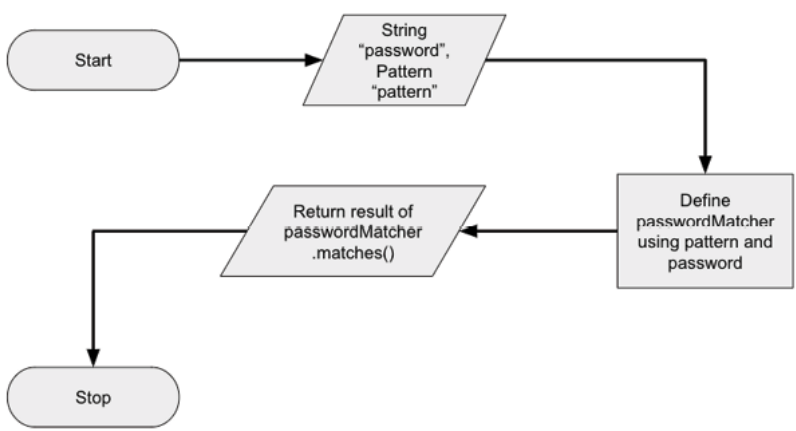

***Figure 6.2**: Flowchart describing the isValidPassword() method*

Java code

The Java code for *Program 6.2*'s **isValidPassword()** method is specified in the following code:

```
Import java.util.regex.Matcher;
Import java.util.regex.Pattern;
Pattern passwordPattern =
    Pattern.compile("(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9]).{12,25}");
private static boolean isValidPassword(String password, Pattern
pattern) {
    Matcher passwordMatcher = pattern.matcher(password);
    return passwordMatcher.matches();
}
```

Note: The code above is meant to be in the same class file as the test code shown at the beginning of this program. While the isValidPassword() method is fine on its own, the passwordPattern should be defined in the class main() method, before calls to the isValidPassword() method. Please see the code addendum for this chapter for more details.

Code explanation

For this problem, we will need to instantiate a new **Pattern** object named **passwordPattern**. It should be instantiated with the result of **Pattern**'s **compile()** static method, with the following regular expression passed as a parameter:

(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9]).{12,25}

The regular expression in the preceding section works like this:

- **(?=.*[a-z])**: This is using parentheses to create a *grouping*. In fact, for this regex, we are going to create multiple groupings, all of which will execute a *positive lookahead* with the question mark/pattern approach. Essentially, this grouping will look ahead at the remaining text and determine if the pattern **[a-z]** (any lowercase character) is a match. If the lookahead operation does indeed find a lowercase character, this grouping will register a positive result.

- **(?=.*[A-Z])**: This grouping is also executing a positive lookahead to verify that there is at least one uppercase character present in the string.
- **(?=.*[0-9])**: This grouping is executing a positive lookahead to verify that there is at least one digit present in the string. Similarly, instead of **(?=.*[0-9])** we could also use **(?=.*\\d)** to determine if a single digit is present.
- **.{12,25}**: Here, we are ensuring that the string is *both* a minimum of 12 characters in length and also no more than 25 characters.

With our regular expression defined, we will write a new method named **isValidPassword**. This method should return a Boolean value, and it should accept a string named **password** and a **Pattern** named **pattern** as its parameters.

Inside the **isValidPassword()** method, we will define a new **Matcher** object named **passwordMatcher**, instantiated with the result from **pattern**'s **matcher()** method, using **password** as a parameter. Finally, we will return the result of **passwordMatcher**'s **matches()** method to the calling program.

Program 6.3: Determine if an IP address is valid

Using regular expressions, build a method to check if an IP address is valid. We can test our method with code similar to the following:

```
String address1 = "127.0.0.1";
String address2 = "192.168.10.101";
String address3 = "034.10.2.7";
String address4 = "33.71.304.88";
String address5 = "192.168.0.256";
String address6 = "192.168.0";
String address7 = ".192.168.0.1";
String address8 = "192.168.0.01";
Pattern ipV4Pattern =
    Pattern.compile("^((25[0-5]|(2[0-4]|1[0-9]|[1-9]|)[0-9])\\.?\
```

```
\b){4}$");
System.out.printf("is %s a valid IPv4 address? %b\n", address1,
    isValidIPv4(address1, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address2,
    isValidIPv4(address2, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address3,
    isValidIPv4(address3, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address4,
    isValidIPv4(address4, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address5,
    isValidIPv4(address5, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address6,
    isValidIPv4(address6, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address7,
    isValidIPv4(address7, ipV4Pattern));
System.out.printf("is %s a valid IPv4 address? %b\n", address8,
    isValidIPv4(address8, ipV4Pattern));
```

This should produce an output similar to the following:

```
is 127.0.0.1 a valid IPv4 address? true
is 192.168.10.101 a valid IPv4 address? true
is 034.10.2.7 a valid IPv4 address? false
is 33.71.304.88 a valid IPv4 address? false
is 192.168.0.256 a valid IPv4 address? false
is 192.168.0 a valid IPv4 address? false
is .192.168.0.1 a valid IPv4 address? false
is 192.168.0.01 a valid IPv4 address? false
```

Algorithm

To solve this problem, we will take the following steps:

1. Define a new **Pattern** object named **ipv4Pattern**, instantiated

with the result of **Pattern**'s **compile()** static method.

a. The **compile()** method should have the following regular expression passed as a parameter: **(?=.*[a-z])(?=.*[A-Z])(?=.*\\d).{12,25}**

2. Build a new method named **isValidIPv4** that accepts a string named **ipAddress** and a **Pattern** named **pattern**. Inside this method, do the following:

a. Define a new **Matcher** object named **addressMatcher** instantiated with the result of **pattern**'s **matcher()** method, while passing **ipAddress** as a parameter.

b. Return the result of **addressMatcher.matches()** to the calling program.

Flowchart

The flowchart for *Program 6.3* can be seen in *Figure 6.3*:

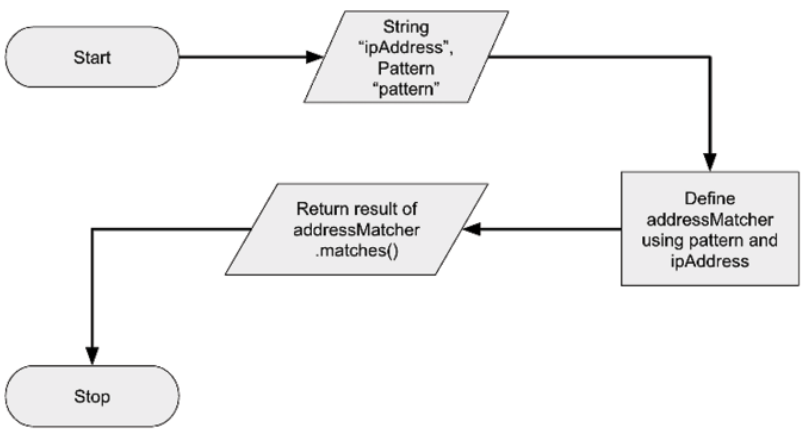

***Figure 6.3**: Flowchart describing the isValidIPv4() method of a stack*

Java code

The Java code for *Program 6.3* is specified in the following section:

```
Import java.util.regex.Matcher;
Import java.util.regex.Pattern;
Pattern ipV4Pattern =
   Pattern.compile("^((25[0-5]|(2[0-4]|1[0-9]|[1-9]|)[0-9])\\.?\\b)
```

```
{4}$");
private static boolean isValidIPv4(String ipAddress, Pattern pattern)
{
    Matcher addressMatcher = pattern.matcher(ipAddress);
    return addressMatcher.matches();
}
```

Note: The code above is meant to be in the same class file as the test code shown at the beginning of this program. While the isValidIPv4() method is fine on its own, the ipV4Pattern should be defined in the class main() method, before calls to the isValidIPv4() method. Please see the code addendum for this chapter for more details.

Code explanation

For this problem, we will need to instantiate a new **Pattern** object named **ipV4Pattern**. It should be instantiated with the result of **Pattern**'s **compile()** static method, with the following regular expression passed as a parameter:

^((25[0-5]|(2[0-4]|1[0-9]|[1-9]|)[0-9])\\.?\\b){4}$

The regular expression in the preceding section works in the following manner:

- **^:** The caret symbol indicates the beginning of the string.
- **((25[0-5]|(2[0-4]|1[0-9]|[1-9]|)[0-9]):** This grouping checks for valid numbers in each octet of the address. As a valid IPv4 octet is 0-255, we first need to check if the octet is between 250-255. The pipe character then functions as an OR operator, and then allows us to check (Nam 2024) if the octet is between 200 and 249, then 100-199, then 10-99, and finally 1-9.
- **\\.?**: This has us optionally check for a dot character. IPv4 address octets are separated by dots, so we should find one in this position, except for the last octet.
- **\\b)**: This character indicates a word boundary, or a logical separation in our string. Again, the addresses are separated by dots, so the word boundary indicator should come after the dot.

- **{4}**: This indicates that we want to perform this check four consecutive times in our string. After all, IPv4 addresses are made up of four octets.
- **$**: The dollar symbol indicates the end of the string.

Note: This regular expression pattern will indicate that 0.0.0.0 is a valid IP address. However, it is important to note (IANA 2023) that the IP range of 0.0.0.0/8 is designated by the Internet Assigned Numbers Authority as restricted.

With our regular expression defined, we will write a new method named **isValidPassword**. This method should return a Boolean value, and it should accept a string named *password* and a **Pattern** named **pattern** as its parameters.

Inside the **isValidPassword()** method, we will define a new **Matcher** object named **passwordMatcher**, instantiated with the result from **pattern**'s **matcher()** method, using **password** as a parameter. Finally, we will return the result of **passwordMatcher**'s **matches()** method to the calling program.

Program 6.4: Compute tollway charges by vehicle

Build a program to compute an appropriate charge for vehicles on a tollway. Motorcycles should be charged $2.00 each, cars $4.00, and trucks are charged based on the number of axles. All other vehicles should be charged $3.00. The code to test our solution should look like the following:

```
Car car1 = new Car("ABM-223", "Ford", "Mustang");
Car car2 = new Car("ILVBEER", "Hyundai", "Ioniq 5");
Car car3 = new Car("TSD-915", "Ford", "Fusion");
Motorcycle cycle1 = new Motorcycle("NP-2112", "BMW", "R1200");
Truck truck1 = new Truck("JLV-925", "Dodge", "Ram 2500", 2,
4000);
Truck truck2 = new Truck("TRO-227", "Volvo", "VNL64760", 5,
46000);
System.out.printf("The toll for the %s %s is $%s\n", car1.getMake(),
```

```
        car1.getModel(), computeToll(car1));
System.out.printf("The toll for the %s %s is $%s\n", car2.getMake(),
        car2.getModel(), computeToll(car2));
System.out.printf("The toll for the %s %s is $%s\n",
truck1.getMake(),
        truck1.getModel(), computeToll(truck1));
System.out.printf("The toll for the %s %s is $%s\n",
truck2.getMake(),
        truck2.getModel(), computeToll(truck2));
System.out.printf("The toll for the %s %s is $%s\n", car3.getMake(),
        car3.getModel(), computeToll(car3));
System.out.printf("The toll for the %s %s is $%s\n",
cycle1.getMake(),
        cycle1.getModel(), computeToll(cycle1));
```

This should produce an output similar to the following:

```
The toll for the Ford Mustang is $4
The toll for the Hyundai Ioniq 5 is $4
The toll for the Dodge Ram 2500 is $4
The toll for the Volvo VNL64760 is $39.00
The toll for the Ford Fusion is $4
The toll for the BMW R1200 is $2
```

Algorithm

To solve this problem, we will take the following steps:

1. Build a new abstract class named **MotorVehicle**.
    a. The **MotorVehicle** class should have three string properties named **licensePlate**, **model**, and **make**.
    b. Each property should also have pairs of corresponding accessor methods (getters and setters).
2. Build a new class named **Truck**. The **Truck** class should extend the **MotorVehicle** class.

a. The **Truck** class should have two integer properties named **axles** and **cargoCapLbs**.

b. Each property should also have pairs of corresponding accessor methods (getters and setters).

c. Build a new public constructor for the **Truck** class. It should accept the following parameters:

i. String **licensePlate**

ii. String **make**

iii. String **model**

iv. Int **axles**

v. Int **cargoCapLbs**

d. The constructor's main task is to set the class properties and the inherited properties with the like-named parameters that were passed in.

i. The properties **licensePlate**, **make**, and **model** should be set using the appropriate setter accessor methods (inherited) from the **MotorVehicle** class.

ii. The properties **axles** and **cargoCapLbs** can be set directly using the **this** keyword.

3. Build a new class named **Car**. The **Car** class should extend the **MotorVehicle** class.

a. The **Car** class does not require any properties of its own.

b. Build a new public constructor for the **Car** class. It should accept the following parameters:

i. String **licensePlate**

ii. String **make**

iii. String **model**

c. The constructor should use the inherited accessor methods to set the appropriate value for each like-named parameter.

4. Build a new class named **Motorcycle**. The **Motorcycle** class should extend the **MotorVehicle** class.

a. The **Motorcycle** class does not require any properties of its own.

b. Build a new public constructor for the **Motorcycle** class. It should accept the following parameters:

i. String **licensePlate**

ii. String **make**

iii. String **model**

c. The constructor should use the inherited accessor methods to set the appropriate value for each like-named parameter.

5. Define a constant in the main program named **TOLL_PER_AXLE** as a new object of the **BigDecimal** class, passing a value of 7.8 to the **BigDecimal** constructor.

6. Build a new method in the main program named **computeToll**. It should return a **BigDecimal** type, and accept a parameter of type **MotorVehicle** named **vehicle**.

a. Begin a switch/case statement with three cases, passing **vehicle** as the lone parameter.

b. The first case should check if **vehicle** is an instance of **Motorcycle m**.

i. If true, return a new **BigDecimal** value of 2 to the calling program.

c. The next case should check if **vehicle** is an instance of **Car c**.

i. If true, return a new **BigDecimal** value of 4 to the calling program.

d. The next case should check if **vehicle** is an instance of **Truck t**.

i. If true, define a local variable named axles, and set it with the result of the **t.getAxles()** method.

ii. If **axles** is > 2, compute the result of **axles** multiplied by **TOLL_PER_AXLE**, and return it to the calling program.

iii. If **axles** is not > 2, then return a new **BigDecimal**

value of 4 to the calling program.

e. For the default case, simply return a new **BigDecimal** value of 3 to the calling program.

Flowchart

The flowchart for *Program 6.4* can be seen in *Figure 6.4:*

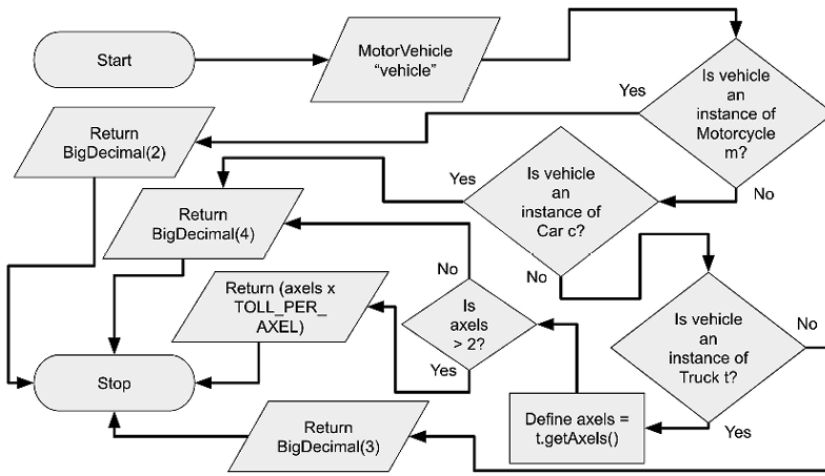

***Figure 6.4**: Flowchart describing the computeToll() method*

Java code

The Java code for *Program 6.4* is specified in the following section:

```
abstract class MotorVehicle {
    private String licensePlate;
    private String model;
    private String make;
    public String getLicensePlate() {
        return licensePlate;
    }
    public void setLicensePlate(String licensePlate) {
        this.licensePlate = licensePlate;
    }
```

```
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
    public String getMake() {
        return make;
    }
    public void setMake(String make) {
        this.make = make;
    }
}
public class Car extends MotorVehicle {
    public Car(String licensePlate, String make, String model) {
        this.setLicensePlate(licensePlate);
        this.setMake(make);
        this.setModel(model);
    }
}
public class Motorcycle extends MotorVehicle {
    public Motorcycle(String licensePlate, String make, String model)
{
        this.setLicensePlate(licensePlate);
        this.setMake(make);
        this.setModel(model);
    }
}
public class Truck extends MotorVehicle {
    private int axles;
```

```
    private int cargoCapLbs;
    public Truck(String licensePlate, String make, String model,
            int axles, int cargoCapLbs) {
        this.setLicensePlate(licensePlate);
        this.setMake(make);
        this.setModel(model);
        this.axles = axles;
        this.cargoCapLbs = cargoCapLbs;
    }
    public int getAxles() {
        return axles;
    }
    public void setAxles(int axles) {
        this.axles = axles;
    }
    public int getCargoCapLbs() {
        return cargoCapLbs;
    }
    public void setCargoCapLbs(int cargoCapLbs) {
        this.cargoCapLbs = cargoCapLbs;
    }
}
import java.math.BigDecimal;
import java.math.RoundingMode;
public final static BigDecimal TOLL_PER_AXLE = new
BigDecimal(7.8);
private static BigDecimal computeToll(MotorVehicle vehicle) {
    switch (vehicle) {
        case Motorcycle m:
            return new BigDecimal(2);
```

```
        case Car c:
            return new BigDecimal(4);
        case Truck t:
            int axles = t.getAxles();
            if (axles > 2) {
                return
TOLL_PER_AXLE.multiply(BigDecimal.valueOf(axles))
                        .setScale(2, RoundingMode.HALF_EVEN);
            } else {
                return new BigDecimal(4);
            }
        default:
            return new BigDecimal(3);
    }
}
```

Note: While the individual classes shown above are meant to be in their own class files, the TOLL_PER_AXLE should be defined in the same class as the main() method. Please see the code addendum for this chapter for more details.

Code explanation

To accurately charge tolls for certain vehicles, we will have to identify them. In Java, we can do this traditionally with the **instanceof** keyword. But as of Java 21, the switch/case construct allows for pattern matching by instance type, so that will be our path forward.

Note: The version of the JDK is important for this program. Switch/case pattern matching was introduced in Java 17 and became a full-fledged feature in Java 21. If the development environment is running a JDK version 17 to 20, preview features will need to be enabled for the compiler.

Before we get started on the logic, we first need to create four new classes. First, we will start with an abstract class named **MotorVehicle**. As an abstract class, **MotorVehicle** will never be

instantiated into an object. But it can be used as a base class by other classes. We will define three string properties inside the **MotorVehicle** class: **licensePlate**, **model**, and **make**. We will also build our getter and setter accessor methods for each property.

Next, we will build a new class named **Truck**. The **Truck** class definition will use the **extends** keyword to inherit from the **MotorVehicle** class. The **Truck** class will also have a public constructor and two integer properties, **axles** and **cargoCapLbs**. The constructor should accept the following parameters: **licensePlate**, **make**, **model**, **axles**, and **CargoCapLbs**. The **licensePlate**, **make**, and **model** inherited properties will need to be set to the values of their like-named parameters using the setter accessor methods from the **MotorVehicle** class, but **axles** and **CargoCapLbs** can be set directly.

Next, we will build a new class named **Car**. The **Car** class definition will use the **extends** keyword to inherit from the **MotorVehicle** class. The **Car** class will not have any parameters, and its public constructor will accept the following parameters: **licensePlate**, **make**, and **model**. Then we will set the like-named inherited properties using their setter accessor methods.

Similarly, we will also build a new class named **Motorcycle**. Other than the names of the class and the constructor, the **Motorcycle** class will be coded exactly the same as the **Car** class. It will extend the **MotorVehicle** class, and its public constructor will accept the parameters: **licensePlate**, **make**, and **model**. The **Motorcycle** constructor should also set the like-named inherited properties using the setter accessor methods.

With all of our supporting classes complete, we can return to our main program. Here, we will import the following libraries:

- **java.math.BigDecimal**
- **java.math.RoundingMode**

Then, we will define a constant named **TOLL_PER_AXLE** of the **BigDecimal** class to an instance of a new **BigDecimal** with a value of 7.8. This is what our toll system will charge large, container-hauling semi-trucks. Smaller passenger trucks will be charged the same as cars.

Note: Floating point types like float and double are inherently inaccurate when expressing the result of arithmetic operations in decimal numbers. Therefore, the BigDecimal class should always be

used when working with monetary calculations.

With all that complete, we can now write our new method named **computeToll**. The **computeToll()** method should return a **BigDecimal**, and accept a **MotorVehicle** object named **vehicle** as a parameter. This will allow our method to process objects that are instances of either **Car**, **Truck**, or **Motorcycle**, since they all inherit from the **MotorVehicle** base class.

Inside the **computeToll()** method, we will build our switch/case statement, passing **vehicle** as a parameter to the **switch** keyword. For the first case, we will check whether or not **vehicle** is an instance of an object **m** of the **Motorcycle** class. The syntax requirements of the case statement require us to specify an instance (**m**), even if we are not going to use it. If the **vehicle** is in fact an instance of the **Motorcycle** class, we will simply return a new **BigDecimal** with a value of 2.

The next case will check whether or not **vehicle** is an instance of an object **c** of the **Car** class. If **vehicle** is in fact an instance of the **Car** class, we will return a new **BigDecimal** with a value of 4.

The last case will check whether or not **vehicle** is an instance of an object **t** of the **Truck** class. If **vehicle** is in fact an instance of the **Truck** class, then we have some work to do. First, we will declare a local variable named **axles** and define it with the result of the **getAxles()** method from the **t** object. If the value of **axles** is > 2, we will use the **multiply()** method from the **TOLL_PER_AXLE BigDecimal** to compute the product of **TOLL_PER_AXLE x axles**.

Once we have that result, we will use the **BigDecimal setScale()** method to limit the response to 2 decimal precision points and enforce a rounding mode of **HALF_EVEN**. This particular decimal rounding mode's behavior (Oracle 2024) statistically reduces the margin of error that can occur over multiple, cumulative calculations. We will then return the resulting **BigDecimal** value to the calling program.

However, if the value of **axles** is not > 2, then we likely have a passenger truck. We can treat those the same as a car, and return a new **BigDecimal** with a value of 4. And if our logic should fall through to the default case, we will then simply return a **BigDecimal** value of 3.

Conclusion

In this chapter, we worked with two different ways to approach pattern matching. We focused heavily on how to use regular expressions to match certain string patterns. We also learned how to use Java's switch/case construct to use pattern matching for different instance types.

While regular expressions are a powerful tool, it is important to remember that there are significant performance trade-offs to be considered for their use. Suppose the matching required is something simple that can be accomplished by using a string method; that is usually the better choice. However, they are a great fit when a string needs to be matched with a complex pattern.

As noted in the preceding section, matching object instance types with the switch/case construct is something that is relatively new to Java. For the Tollway example, an alternative would be to write a long if-then-else statement using the instanceof keyword. The switch/case pattern matching feature allows us to express that logic much more concisely.

In the next chapter, we will take another look at data structures. However, this time our focus will be on more complex data structures like linked lists, binary trees, and heaps. We will also demonstrate how to use these structures to solve real-world problems.

Points to remember

Have a look at some key points to remember:

- Regular expression patterns should be compiled only once and used many times.
- Character sets are encapsulated inside brackets.
- Groupings are encapsulated inside parentheses.
- The //d indicates a digit or numeric character, and is similar to specifying a character set of [0-9].
- A lookahead is a way of checking the rest of the string.

- An abstract class is one that will never be instantiated into an object.
- Instead of a float or a double, the BigDecimal class should be used when working with currency.

Questions

Answer the following questions to check your understanding:

1. The Matcher class has other methods available for determining a match. Why are we using the matches() method?
2. What is the metacharacter that is used to indicate the end of a string?
3. How could we match on a string while enforcing both a minimum and a maximum number of characters?
4. What is the main benefit of the HALF_EVEN rounding mode?
5. How would we write a regular expression to match a vehicle license plate pattern of three characters, followed by a dash, and then three numbers?
6. How would we write a switch/case statement to see if an object number was of an Integer, Double, or Float?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Section 3:
Challenging Problems

Chapter 7

Complex Data Structures

Introduction

In *Chapter 5, Basic Data Structures*, we talked about basic data structures and worked through some examples showing how to use them to solve real-world problems. However, those of us who have had a data structure course at the university level have probably noticed that we have barely scratched the surface of the topic.

Here, we will discuss in detail the topic of data structures. We will introduce new structures, show how they work, and demonstrate how to apply them to solve more challenging problems. Additionally, we will start by revisiting a more advanced version of a structure that we looked at earlier.

Structure

This chapter will cover the following programs:

- Program 7.1: Build a linked list for browser history

- Program 7.2: Build a binary tree to store fruit data
- Program 7.3: Build a heap for a medical queue

Objectives

The main idea of this chapter is to take our understanding of data structures to the next level. We will look at some new data structures and also revisit some from the previous chapter(s). We will take another look at a linked list, but with a bit of a twist. Additionally, we will explore new structures like binary trees and heaps. The goal is that by performing simple exercises with these more advanced data structures, we will not only start to understand them but also see them in common, modern applications.

Program 7.1: Build a linked list for browser history

Build a two-way linked list as an underlying structure for a web browser's navigation history. While similar to the linked list that we built in *Chapter 5*, *Basic Data Structures,* a two-way linked list allows its items to be traversed in both directions. Thus, it makes a great underlying structure for browser history, as web browsers allow you to traverse your history both forward and backward.

We can test our two-way linked list with code similar to the following:

```
TwoWayLinkedList browserHistory = new TwoWayLinkedList();

HistoryItem historyItem1 = new HistoryItem("https://
www.stackoverflow.com/users/1054558/aaron");

HistoryItem historyItem2 = new HistoryItem("https://
www.stackoverflow.com/questions/tagged/cassandra");

HistoryItem historyItem3 = new HistoryItem("https://
www.stackoverflow.com/questions/35708118/where-and-order-by-
clauses-in-cassandra-cql");

HistoryItem historyItem4 = new HistoryItem("https://
www.datastax.com/blog/we-shall-have-order");

HistoryItem historyItem5 = new HistoryItem("https://
```

```
www.datastax.com");
HistoryItem historyItem6 = new HistoryItem("https://
www.datastax.com/products/datastax-astra");
browserHistory.addHistoryItem(historyItem1);
browserHistory.addHistoryItem(historyItem2);
browserHistory.addHistoryItem(historyItem3);
browserHistory.addHistoryItem(historyItem4);
browserHistory.addHistoryItem(historyItem5);
browserHistory.addHistoryItem(historyItem6);
traverseHistoryForward(browserHistory);
traverseHistoryBackward(browserHistory);
```

This should produce an output similar to the following:

```
Traversing browser history forward:
https://www.stackoverflow.com/users/1054558/aaron
https://www.stackoverflow.com/questions/tagged/cassandra
https://www.stackoverflow.com/questions/35708118/where-
and-order-by-clauses-in-cassandra-cql
https://www.datastax.com/blog/we-shall-have-order
https://www.datastax.com
https://www.datastax.com/products/datastax-astra
Traversing browser history backward:
https://www.datastax.com/products/datastax-astra
https://www.datastax.com
https://www.datastax.com/blog/we-shall-have-order
https://www.stackoverflow.com/questions/35708118/where-
and-order-by-clauses-in-cassandra-cql
https://www.stackoverflow.com/questions/tagged/cassandra
https://www.stackoverflow.com/users/1054558/aaron
```

Algorithm

Before diving into the algorithm, let us consider what components

are necessary for a two-way linked list, and what functions they should be able to perform. We will need to keep track of each individual browser history item, which needs a URL and structures to link to its neighbors.

What about managing a collection of these historical items? We will need a class that holds the head and tail, and keeps a count of the number of items.

What are the primary behaviors that we need to support? We will need methods to handle tasks like adding a new item and traversing forward and backward.

These are questions that we should be asking before writing any code.

To solve this problem, we will take the following steps:

1. Build a new class named **HistoryItem**. It should have:
    a. The following class properties:
        i. A **UUID** named **id**.
        ii. A **string** named **url**.
        iii. A self-referential **HistoryItem** object named **nextHistoryItem**.
        iv. A self-referential **HistoryItem** object named **previousHistoryItem**.
    b. A constructor that accepts a string named **url** as a parameter. Inside the constructor:
        i. Sets the **id** class property to a new, random **UUID**.
        ii. Sets the **url** class property to the **url** parameter.
    c. A getter accessor method for **id** named **getId**.
    d. Both getter and setter accessor methods for the **url**, **nextHistoryItem**, and **previousHistoryItem** properties.
2. Build a new class named **TwoWayLinkedList**. It does not require a constructor and should have:
    a. The following class properties:
        i. A **HistoryItem** object named **firstHistoryItem**.
        ii. A **HistoryItem** object named **lastHistoryItem**.

iii. An integer named **historyItemCount**, initialized to 0.

iv. Getter accessor methods for all properties.

b. A method named **addHistoryItem** that accepts a **HistoryItem** object named **histItem**. Inside this method, do the following:

i. First, check if the value of **historyItemCount** is greater than 0.

ii. If **historyItemCount** is not greater than 0, simply set **firstHistoryItem** to **histItem**.

iii. Outside of the if statement, set **lastHistoryItem** to **histItem**.

iv. Increment **historyItemCount**.

3. Build a new method in our main program named **traverseHistoryForward**. It should accept a **TwoWayLinkedList** object named **browserHistory** as a parameter. Inside the method, do the following:

a. Print a formatted statement indicating that it is **Traversing browser history forward over %d items** while passing the result of **browserHistory**'s **getHistoryItemCount()** method as a parameter.

b. Define a local **HistoryItem** object named current, initialized with the result of **browserHistory**'s **getFirstHistoryItem()** method.

c. Build a while loop that iterates as long as current is not equal to null. Inside the loop:

i. Print the result of **current**'s **getUrl()** method.

ii. Set **current** equal to the result of **current**'s **getNextHistoryItem()** method.

4. Build a new method in our main program named **traverseHistoryBackward**. It should accept a **TwoWayLinkedList** object named **browserHistory** as a parameter. Inside the method:

a. Print a formatted statement indicating that it is **Traversing browser history backward over %d**

items while passing the result of **browserHistory**'s **getHistoryItemCount()** method as a parameter.

b. Define a local **HistoryItem** object named current, initialized with the result of **browserHistory**'s **getLastHistoryItem()** method.

c. Build a while loop that iterates as long as **current** is not equal to null. Inside the loop:

i. Print the result of **current**'s **getUrl()** method.

ii. Set **current** equal to the result of **current**'s **getPreviousHistoryItem()** method.

Flowchart

The flowcharts for *Program 7.1* can be seen in the following *Figures 7.1*, *7.2*, and *7.3*:

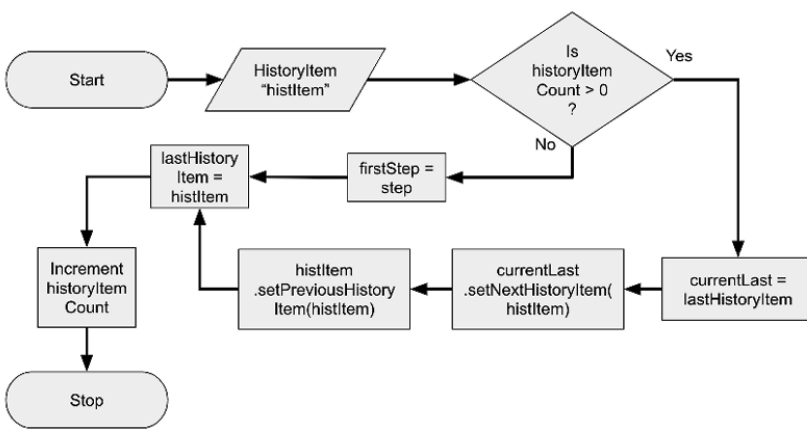

***Figure 7.1**: Flowchart describing TwoWayLinkedList's addHistoryItem() method*

The following figure demonstrates how a two-way linked list is traversed forward to iterate through browser history and display each visited URL:

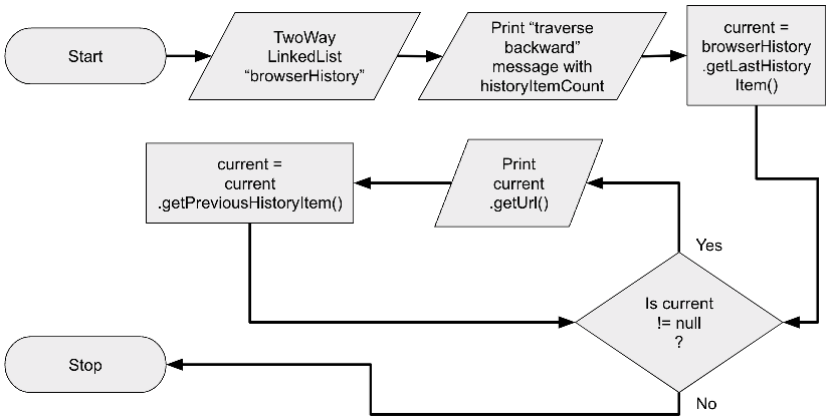

***Figure 7.2**: Flowchart describing the traverseHistoryForward() method*

The following figure illustrates backward traversal of a two-way linked list, showing how browser history is iterated in reverse to access previously visited URLs:

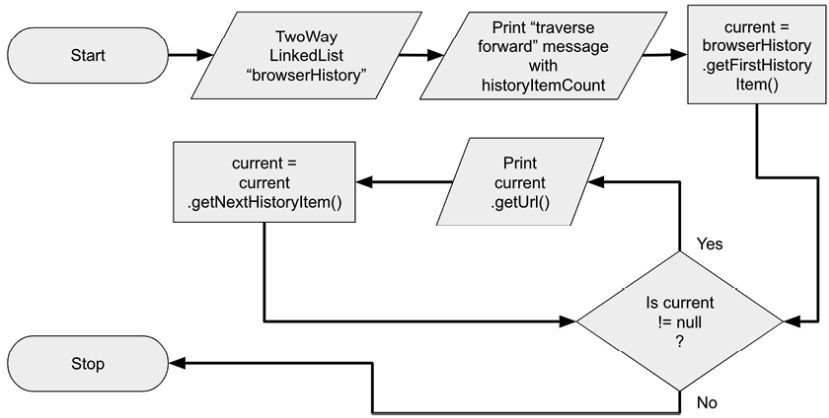

***Figure 7.3**: Flowchart describing the traverseHistoryBackward() method*

Java code

Let us have a look at the Java code for *Program 7.1*. Here is the code for the **HistoryItem** class:

```
import java.util.UUID;
public class HistoryItem {
```

```
    private UUID id;
    private String url;
    private HistoryItem nextHistoryItem;
    private HistoryItem previousHistoryItem;
    public HistoryItem(String url) {
        this.id = UUID.randomUUID();
        this.url = url;
    }
    public UUID getId() {
        return id;
    }
    public String getUrl() {
        return url;
    }
    public void setUrl(String url) {
        this.url = url;
    }
    public HistoryItem getNextHistoryItem() {
        return nextHistoryItem;
    }
    public void setNextHistoryItem(HistoryItem nextHistoryItem) {
        this.nextHistoryItem = nextHistoryItem;
    }
    public HistoryItem getPreviousHistoryItem() {
        return previousHistoryItem;
    }
    public void setPreviousHistoryItem(HistoryItem
previousHistoryItem) {
        this.previousHistoryItem = previousHistoryItem;
    }
```

```
}
```

Next, we will look at the code for the **TwoWayLinkedList** class, which can be seen here:

```
public class TwoWayLinkedList {
    private HistoryItem firstHistoryItem;
    private HistoryItem lastHistoryItem;
    private int historyItemCount = 0;
    public void addHistoryItem(HistoryItem histItem) {
        if (historyItemCount > 0) {
            HistoryItem currentLast = lastHistoryItem;
            currentLast.setNextHistoryItem(histItem);
            histItem.setPreviousHistoryItem(currentLast);
        } else {
            firstHistoryItem = histItem;
        }
        lastHistoryItem = histItem;
        historyItemCount++;
    }
    public HistoryItem getFirstHistoryItem() {
        return firstHistoryItem;
    }
    public HistoryItem getLastHistoryItem() {
        return lastHistoryItem;
    }
    public int getHistoryItemCount() {
        return this.historyItemCount;
    }
}
```

Additionally, the following methods **traverseHistoryForward()** and **traverseHistoryBackward()** should be in our primary class (we can

name it **Chapter7.java**, and it should have a **main** method). The code for these methods can be seen as follows:

```
private static void traverseHistoryForward(TwoWayLinkedList
browserHistory) {
    System.out.printf("\nTraversing browser history forward over %d
items:\n",
            browserHistory.getHistoryItemCount());
    HistoryItem current = browserHistory.getFirstHistoryItem();
    while (current != null) {
        System.out.printf("%s\n", current.getUrl());
        current = current.getNextHistoryItem();
    }
}
private static void traverseHistoryBackward(TwoWayLinkedList
browserHistory) {
    System.out.printf("\nTraversing browser history backward over
%d items:\n",
            browserHistory.getHistoryItemCount());
    HistoryItem current = browserHistory.getLastHistoryItem();
    while (current != null) {
        System.out.printf("%s\n", current.getUrl());
        current = current.getPreviousHistoryItem();
    }
}
```

Code explanation

For this problem, we will first need to create two classes: **HistoryItem** and **TwoWayLinkedList**. Each website that is visited inside the (theoretical) browser will be instantiated as a **HistoryItem** object. Our **HistoryItem** class will have four properties: a **UUID** named **id** (do not forget to import **java.util.UUID**), a string named **url**, and two self-referential objects of the **HistoryItem** class named **nextHistoryItem** and **previousHistoryItem**.

The public constructor for **HistoryItem** will accept a string parameter named **url**. Inside the constructors it will use that parameter to set the **url** class property, and the **id** class property will be instantiated as a new, random UUID. We should also build a getter accessor method for the **id** property, as well as getter and setter accessors for the remaining properties (**url**, **nextHistoryItem**, and **previousHistoryItem**).

Note: As users frequently visit the same web pages, the UUID will help us ensure uniqueness among the HistoryItem objects. The UUID generation is also handled by the constructor, so we should not need a setId() accessor method.

Next, we will create a new class named **TwoWayLinkedList**. While there will be some similarities with the **LinkedList** class that we wrote in *Chapter 5, Basic Data Structures,* this class will be different as it will allow both forward and backward traversals of the items. If we recall, the **LinkedList** class only allowed item traversals in a forward direction, classifying it as a *one-way* linked list. Our **TwoWayLinkedList** class should have three class properties: two **HistoryItem** objects named **firstHistoryItem** and **lastHistoryItem**, as well as an integer named **historyItemCount**. The default, parameter-less constructor should be fine for our purposes, so there will be no need to create one.

Inside the **TwoWayLinkedList** class, we will define a method named **addHistoryItem**. This method should accept a **HistoryItem** object named **histItem** as its only parameter. Inside this method, we will first check if **historyItemCount** is > 0. If true, we will create a new **HistoryItem** reference named **currentLast** to make it easier for us to swap **HistoryItem** objects as needed. We will then use the **setNextHistoryItem()** method from the **currentLast** object (with **histItem** as a parameter), to set the new item as the *next* item on the current last item. Then, we will use the **setPreviousHistoryItem()** method from **histItem** (with **currentLast** as a parameter) to set the current last item as the new item's *previous* item.

Otherwise, if **historyItemCount** is not > 0 (list is empty), we will simply set the **firstHistoryItem** property to **histItem**. As all items are added to this list from the back, we will set **lastHistoryItem** to **histItem** on every call to this method. Finally, we will increment

historyItemCount before returning to the calling method or program.

We will create getter accessor methods for all of the properties. The assignment of the class properties will be handled by the **add()** method, so we will not need to define any setter or accessor methods.

With the **HistoryItem** and **TwoWayLinkedList** classes created, we can focus our attention on writing two methods in our main program, **traverseHistoryForward** and **traverseHistoryBackward**. We will start with the **traverseHistoryForward()** method, which will accept a **TwoWayLinkedList** object named **browserHistory** as its lone property. This method will begin by printing the message **Traversing browser history forward over %d items**, where the value returned from **browserHistory**'s **getHistoryItemCount()** method will supply the number of items. We will instantiate a new **HistoryItem** object named **current**, with the result of **browserHistory**'s **getFirstHistoryItem()** method. We will then write a while loop that will iterate so long as current is not equal to null. Inside the loop, we will print the result of **current**'s **getUrl()** method, before setting current equal to the result of current's **getNextHistoryItem()** method. Once we reach the last item in the list, the **getNextHistoryItem()** method will return null, and the loop will break.

The **traverseHistoryBackward()** method will be very similar. It will also accept a **TwoWayLinkedList** object named **browserHistory**, and will also begin by printing the message **Traversing browser history backward over %d items**, using **browserHistory**'s **getFirstHistoryItem()** method. However, this method will differ in that we will instantiate a new **HistoryItem** object named current with the result of **browserHistory**'s **getHistoryItemCount()** method. Similarly, we will write a while loop to iterate as long as current is not equal to null. Inside the loop, we will print the result of **current**'s **getUrl()** method, before setting current equal to the result of **current**'s **getPreviousHistoryItem()** method. Once we reach the first item in the list, the **getPreviousHistoryItem()** method will return null, and the loop will break.

Program 7.2: Build a binary tree to store fruit data

Build a binary tree as an underlying structure to store data for a simple fruit application. We can test it with code similar to the following:

```
Node fruit1 = new Node(0, "apple");
Node fruit2 = new Node(1, "banana");
Node fruit3 = new Node(2, "orange");
Node fruit4 = new Node(3, "grape");
Node fruit5 = new Node(4, "grapefruit");
BinaryTree tree = new BinaryTree();
tree.insert(fruit1);
tree.insert(fruit2);
tree.insert(fruit3);
tree.insert(fruit4);
tree.insert(fruit5);
// find id 1 - banana
System.out.printf("Find by key %d = %s\n", 1,
tree.find(1).getName());
// find id 2 - orange
System.out.printf("Find by key %d = %s\n", 2,
tree.find(2).getName());
// find id 7
System.out.printf("Find by key %d = %s\n", 7,
tree.find(7).getName());
// find id 0 - apple
System.out.printf("Find by key %d = %s\n", 0,
tree.find(0).getName());
```

This should produce an output similar to the following:

```
Find by key 1 = banana
Find by key 2 = orange
```

Find by key 7 = NOT FOUND

Find by key 0 = apple

Algorithm

How should we go about breaking down this problem? Well, we will need a class to keep track of each node in the tree. A node will have a key and a name. It will also need to be able to reference its parent and children.

Additionally, we are also going to need a class for the binary tree itself. This class will need to keep track of the *root* node and require methods to handle node operations (insert, delete, traverse).

To solve this problem, we will take the following steps:

1. Define a new class named **Node**. It should have:
    a. The following class properties:
        i. An integer named **key**.
        ii. A string named **name**.
        iii. A self-referential Node object named **leftChild**.
        iv. A self-referential Node object named **rightChild**.
    b. A public constructor that accepts an integer parameter named **key** and a string parameter named **name**. Inside the constructor:
        i. Set the **key** class property to the value of the **key** parameter.
        ii. Set the **name** class property to the value of the **name** parameter.
    c. Getter and setter accessor methods for the **key** and **name** properties.
    d. A public method named **insert** that accepts a **Node** parameter named **newNode**. This method should implement an if statement to:
        i. Check if the value returned by **newNode**'s **getKey()** method is < the value of the **key** class property. If **true**, check if **leftChild** is equal to null. If true, then set **leftChild** = **newNode**. Otherwise, call

leftChild's **insert()** method while passing **newNode** as a parameter.

ii. Check if the value returned by **newNode**'s **getKey()** method is > the value of the **key** class property. If true, check if **rightChild** is equal to null. If that is true, set **rightChild** = **newNode**. Otherwise, call **rightChild**'s **insert()** method while passing **newNode** as a parameter.

iii. Otherwise, it is assumed that the value returned by **newNode**'s **getKey()** method is equal to the value of the **key** class property. In this case, set the **name** class property to the result of **newNode**'s **getName()** method.

e. A public method named **findByKey** that returns a **Node** object and accepts an integer named **lookupKey** as a parameter. This method should implement an if statement to:

i. Check if the value of **lookupKey** is < the value of the **key** property. If true, then check if **leftChild** is equal to null. If true, then return a new **Node** object passing values of -1 and **"NOT FOUND"** as constructor parameters. Else, return the result of **leftChild**'s **findByKey()** method.

ii. Check if the value of **lookupKey** is > the value of the **key** property. If true, then check if **rightChild** is equal to null. If true, return a new **Node** object passing values of -1 and "**NOT FOUND**" as constructor parameters. Else, return the result of **rightChild**'s **findByKey()** method.

iii. Otherwise, it is assumed that the value of **lookupKey** is equal to the value of the **key** class property. In this case, we will simply return the current **Node** using the **this** keyword.

2. Define a new class named **BinaryTree**. It should have:

a. A class property object of the **Node** class named **root**.

b. A public method named **find**, that returns a **Node** object

and accepts an integer parameter named **key**. This method simply returns the result of **root**'s **findByKey()** method while passing **key** as a parameter.

c. A public method named **insert**, which accepts a **Node** object named **newNode** as a parameter. Inside this method:

i. Check if the **root** class property is equal to null. If true, the tree is empty, so set **root** to **newNode**.

ii. Else, call **root**'s **insert()** method while passing **newNode** as a parameter.

Flowchart

The flowcharts for *Program 7.2* can be seen in the following *Figures 7.4* and *7.5*:

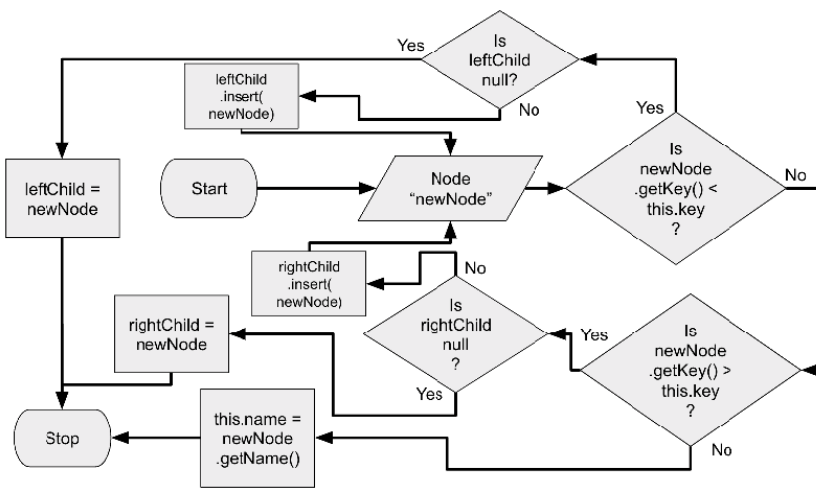

***Figure 7.4**: Flowchart describing the Node class's insert() method*

The following figure depicts the recursive search logic in a binary search tree, illustrating how a lookup key is compared and traversed to locate the matching node or determine that it is not found:

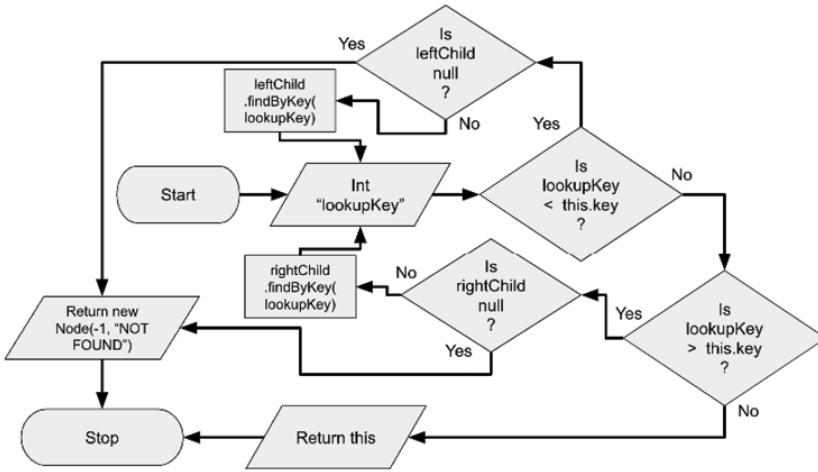

***Figure 7.5**: Flowchart describing the Node class's findByKey() method*

Java code

The Java code for *Program 7.2*'s **Node** class can be found as follows:

```
public class Node {
    private int key;
    private String name;
    private Node leftChild;
    private Node rightChild;
    public Node(int key, String name) {
        this.key = key;
        this.name = name;
    }
    public void insert(Node newNode) {
        if (newNode.getKey() < this.key) {
            if (leftChild == null) {
                leftChild = newNode;
            } else {
                leftChild.insert(newNode);
            }
```

```
    } else if (newNode.getKey() > this.key) {
        if (rightChild == null) {
            rightChild = newNode;
        } else {
            rightChild.insert(newNode);
        }
    } else {
        this.name = newNode.getName();
    }
}
public Node findByKey(int lookupKey) {
    if (lookupKey < this.key) {
        if (leftChild == null) {
            return new Node(-1,"NOT FOUND");
        } else {
            return leftChild.findByKey(lookupKey);
        }
    } else if (lookupKey > this.key ) {
        if (rightChild == null) {
            return new Node(-1,"NOT FOUND");
        } else {
            return rightChild.findByKey(lookupKey);
        }
    } else {
        return this;
    }
}
public int getKey() {
    return key;
}
```

```
    public void setKey(int key) {
        this.key = key;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

The code for the **BinaryTree** class is shown as follows:

```
public class BinaryTree {
    private Node root;
    public void insert(Node newNode) {
        if (root == null) {
            root = newNode;
        } else {
            root.insert(newNode);
        }
    }
    public Node find(int key) {
        return root.findByKey(key);
    }
}
```

Code explanation

To solve this problem, we will implement a binary tree structure. A binary tree consists of nodes connected to each other in a parent-to-children relationship, which gives the appearance of a tree when diagrammed. The first node in a binary tree is known as the *root* node, and it (like all other nodes in the tree) has no more than two

child nodes.

Each node is arranged such that the value of its key is greater than that of its left child and less than that of its right child. This makes key-based lookups in a binary tree very efficient, as each level reduces the remaining set of possible values by half. A sample binary tree can be seen in the following *Figure 7.6*:

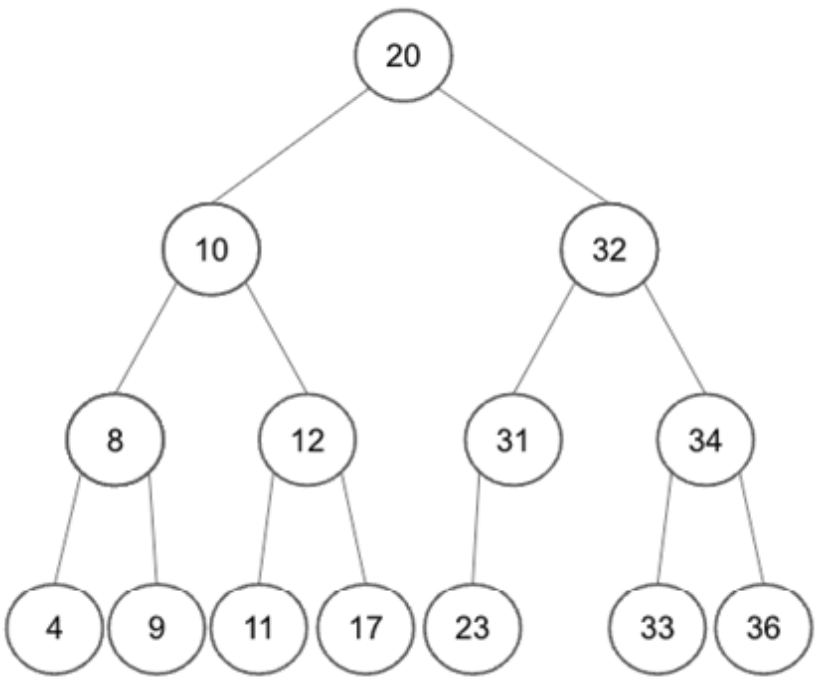

***Figure 7.6**: A sample binary tree holding jersey numbers for a hockey team, where 20 is the root node*

Note: Binary trees are not to be confused with the B-Tree indexing used by database systems. The latter refers to a structure known as a balanced tree.

Getting back to the primary issue at hand, we will begin by building a new class named **Node**. This class should have private properties defined for an integer named **key** and a string named **name**. It should also have two self-referential property objects of type **Node** named **leftChild** and **rightChild**. The **Node** class should have a

public constructor that accepts an integer parameter named **key** and a string parameter named **name**, as well as getter and setter accessor methods for those two properties. The constructor should use these parameters to define the like-named class properties.

The **Node** class should have a public method named **insert** that accepts a **Node** object named **newNode** as a parameter. This method should use an if check to assess whether or not the value returned by **newNode**'s **getKey()** method is < the **key** class property. If that is true, another nested if statement should be used to see if the **leftChild** property is null. If it is, **leftChild** should then be set to **newNode**. If **leftChild** has a value (is not null), then we should call **leftChild**'s **insert()** method while passing **newNode** as a parameter.

The next part of the if statement should assess whether or not the value returned by **newNode**'s **getKey()** method is > the **key** class property. If that is true, another nested if statement should be used to see if the **rightChild** property is null. If it is, **rightChild** should then be set to **newNode**. If **rightChild** has a value (is not null), then we should call **rightChild**'s **insert()** method while passing **newNode** as a parameter.

Finally, we should have an else construct which assumes that the value returned by **newNode**'s **getKey()** method is equal to the **key** property value. In that case, we should assume that we are overwriting all data stored at this node and set the **name** class property to the result of **newNode**'s **getName()** method.

Next, we will need to write a public method for the **Node** class named **findByKey**. This method should return a **Node** object and accept an integer named **lookupKey** as a parameter. This method will use an if check to see if the value of **lookupKey** is < the value of the **key** property. If this is the case, we should next check if **leftChild** is null, and return a new **Node** with a key of -1 and a **name** of **"NOT FOUND"**. If **leftChild** does exist, then we should return the result of **leftChild**'s **findByKey()** method while passing **lookupKey** as a parameter.

The next part of the if check should assess whether or not the value of **lookupKey** is > the value of the **key** property. If this is the case, we should next check if **rightChild** is null, and return a new **Node** with a **key** of -1 and a **name** of "**NOT FOUND**". If **rightChild** does

exist, then we should return the result of **rightChild**'s **findByKey()** method while passing **lookupKey** as a parameter.

Note: In this case, we are defining a new "NOT FOUND" node with a key of -1 to indicate that a node could not be found. While this simplifies our error handling, we could also simply return null, and then also check for null in the calling program.

Finally, we should have an else construct that assumes that the value of **lookupKey** is equal to the **key** property value. In that case, we should simply return the current **Node** instance to the calling program using the **this** keyword.

With the **Node** class built, we can write the **BinaryTree** class. This class should have a single private property object of the **Node** class named **root**.

Inside the **BinaryTree** class, we will write a new method named **insert**, which should accept a **Node** named **newNode** as a parameter. If the **root** property is null, then we should set **root** equal to **newNode**. Otherwise, we should call **root**'s **insert()** method while passing **newNode** as a parameter.

We should also write a new method named **find**, which returns a **Node** object and accepts an integer named **key** as a parameter. This method should simply return the result of **root**'s **findByKey()** method while passing **key** as a parameter.

With these two classes in place, our testing code from the beginning of this section should produce the desired output.

Program 7.3: Build a heap for a medical queue

Using Java's delivered **PriorityQueue** library, build a heap structure to support priority queuing for medical patients. We should be able to test it out with code similar to the following:

```
Patient[] patients = new Patient[5];
patients[0] = new Patient("Paul",3);
patients[1] = new Patient("Jenny",2);
patients[2] = new Patient("Tom",8);
patients[3] = new Patient("Jamie",1);
```

```
patients[4] = new Patient("Tiffany",11);
MedicalHeapJava medQueue = new MedicalHeapJava(patients);
System.out.printf("Patient priority: %s\n",medQueue);
medQueue.push(new Patient("Robert",4));
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
medQueue.push(new Patient("Ron",5));
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
System.out.printf("Now treating patient %s.\n",
medQueue.pop().getName());
System.out.printf("Patient priority: %s\n",medQueue);
```

This should produce an output similar to the following:

```
Patient priority: [ Jamie:1, Jenny:2, Tom:8, Paul:3, Tiffany:11 ]
Patient priority: [ Jamie:1, Jenny:2, Robert:4, Paul:3,
Tiffany:11, Tom:8 ]
Now treating patient Jamie.
Patient priority: [ Jenny:2, Paul:3, Robert:4, Tom:8, Tiffany:11
]
```

Patient priority: [Jenny:2, Paul:3, Robert:4, Tom:8, Tiffany:11, Ron:5]

Now treating patient Jenny.

Patient priority: [Paul:3, Ron:5, Robert:4, Tom:8, Tiffany:11]

Now treating patient Paul.

Patient priority: [Robert:4, Ron:5, Tiffany:11, Tom:8]

Now treating patient Robert.

Patient priority: [Ron:5, Tom:8, Tiffany:11]

Now treating patient Ron.

Patient priority: [Tom:8, Tiffany:11]

Now treating patient Tom.

Patient priority: [Tiffany:11]

Algorithm

What kinds of components will we need to solve this problem? This particular heap will keep track of medical patients, so we will need a class to keep track of patient names and priorities. We will also need a class from which to manage the heap and its operations, like *push* and *pop*.

To solve this problem, we will take the following steps:

1. Define a new class named **Patient**.
    a. It should have the following properties, defined with getter and setter accessor methods:
        i. A string named **name**.
        ii. An integer named **priority**.
    b. It should have a single, public constructor that accepts a string named **name** and an integer named **priority** as parameters. The parameters should be used to initialize the like-named class properties.
2. Define a new class named **PatientComparator**. It should:
    a. Import the **java.util.Comparator** library.
    b. Implement the **Comparator** interface, with a type of **Patient**.

c. Contain a single method (prefaced with the **@Override** annotation) named **compare**. This **compare()** method should return an integer, and should accept two **Patient** parameters named **p1** and **p2**, respectively.

i. The method should use an if construct to determine if the value returned from **p1**'s **getPriority()** method is greater than the value returned from **p2**'s **getPriority()** method. If it is, then a 1 should be returned to the calling program.

ii. If the value returned from **p1**'s **getPriority()** method is less than the value returned from **p2**'s **getPriority()** method, then a –1 should be returned to the calling program.

iii. If the priorities of **p1** and **p2** are equal, then a 0 should be returned.

3. Define a new class named **MedicalHeapJava**. It should:

a. Import the **java.util.PriorityQueue** library.

b. Instantiate a **PriorityQueue** property typed with the **Patient** class and named **patientHeap**.

c. Have two public constructors:

i. The first constructor does not accept any parameters. It simply instantiates the **patientHeap** property with a new **PriorityQueue** of the same type **Patient**, while passing a new, unnamed **PatientComparator** object as its only parameter.

ii. The second constructor should accept an array of **Patient** named **patientArray**. It should also instantiate the **patientHeap** property with a new **PriorityQueue** of the same type **Patient**, while passing a new, unnamed **PatientComparator** object as its only parameter. It should use a for loop to iterate through **patientArray** and use the **patientHeap**'s **add()** method to push the **patient** objects onto the heap.

d. Have a method named **push**. It should not return a value,

and should accept a **Patient** named **patient** as a parameter. This method should push the **patient** parameter onto the heap using **patientHeap**'s **add()** method.

e. Have a method named **pop** that does not accept any parameters and returns an object of type **Patient**. This method should simply return the result of **patientHeap**'s **poll()** method to the calling program.

f. Have a method named **toString** that overrides the delivered **toString()** method, does not accept any parameters, and returns a **String**. This method should:

   i. Instantiate a new **StringBuilder** object named **returnVal**.

   ii. Define a Boolean variable named **first**, with an initial value of **true**.

   iii. Use a for loop that iterates through the **patient** objects in **patientHeap**. It should take the result of each **patient**'s **getName()** and **getPriority()** method, and append them so that they appear as a comma-delimited list inside of two brackets.

   iv. Return the result of **returnVal**'s **toString()** method.

Flowchart

The flowcharts for *Program 7.3*'s **PatientComparator**'s **compare()** method, **MedicalHeapJava**'s **toString()** method, and the **PriorityQueue**'s "**min Heapify**" algorithm can be seen in *Figures 7.7*, *7.8,* and *7.9*, respectively:

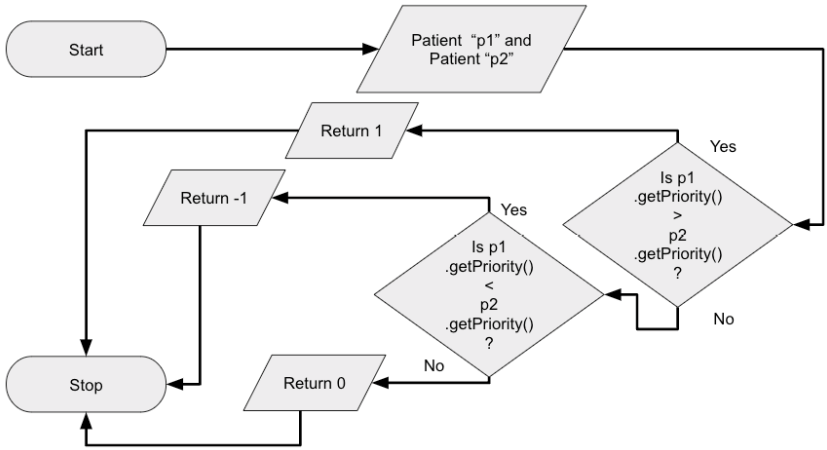

***Figure 7.7**: Flowchart describing the PatientComparator's compare() method*

The following figure illustrates the step-by-step process of iterating through patient records to build a formatted string output while handling separators and ordering logic:

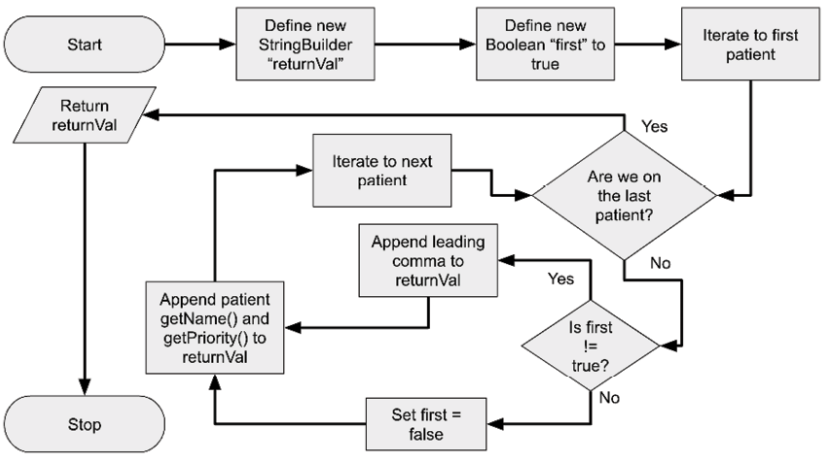

***Figure 7.8**: Flowchart describing the MedicalHeapJava's toString() method*

The following figure depicts the **PriorityQueue**'s internal algorithm in a flowchart form

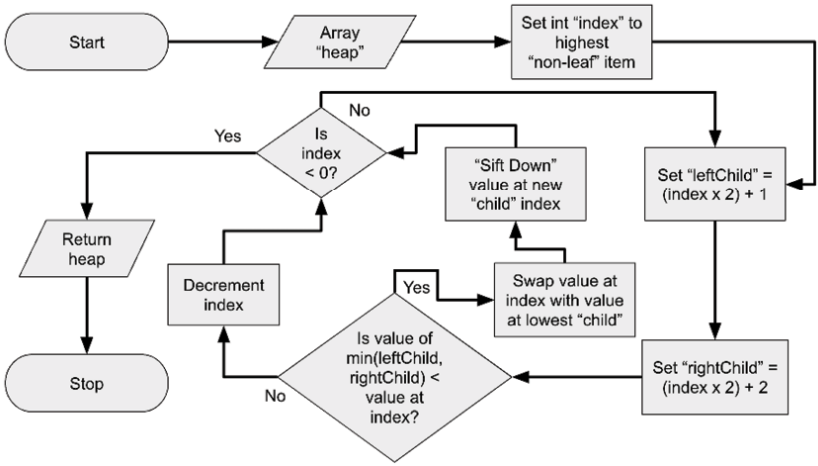
Start
Array "heap"
Set int "index" to highest "non-leaf" item
No
Yes
Is index < 0?
"Sift Down" value at new "child" index
Set "leftChild" = (index x 2) + 1
Return heap
Decrement index
Yes
Swap value at index with value at lowest "child"
Stop
Is value of min(leftChild, rightChild) < value at index?
Set "rightChild" = (index x 2) + 2
No

***Figure 7.9**: Flowchart describing the PriorityQueue's internal min Heapify algorithm*

Java code

The Java code for *Program 7.3*'s **Patient** class is as follows:

```
public class Patient {
    private String name;
    private int priority;
    public Patient(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getPriority() {
        return priority;
```

```
    }
    public void setPriority(int priority) {
        this.priority = priority;
    }
}
```

The Java code for *Program 7.3*'s **PatientComparator** class is as follows:

```
import java.util.Comparator;
public class PatientComparator implements Comparator<Patient>{
    @Override
    public int compare(Patient p1, Patient p2) {
        if (p1.getPriority() > p2.getPriority()) {
            return 1;
        } else if (p1.getPriority() < p2.getPriority()) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

The Java code for *Program 7.3*'s **MedicalHeapJava** class is shown as follows:

```
import java.util.PriorityQueue;
public class MedicalHeapJava {
    PriorityQueue<Patient> patientHeap;
    public MedicalHeapJava(Patient[] patientArray) {
        patientHeap = new PriorityQueue<>(new
PatientComparator());
        for (Patient patient : patientArray) {
            patientHeap.add(patient);
```

```
    }
}
public void push(Patient patient) {
    patientHeap.add(patient);
}
public Patient pop() {
    return patientHeap.poll();
}
public String toString() {
    StringBuilder returnVal = new StringBuilder("[ ");
    boolean first = true;
    for (Patient patient : patientHeap) {
        if (!first) {
            returnVal.append(", ");
        } else {
            first = false;
        }
        returnVal.append(patient.getName());
        returnVal.append(":");
        returnVal.append(patient.getPriority());
    }
    returnVal.append(" ]");
    return returnVal.toString();
}
```

Code explanation

To solve the problem of a priority-based queue for emergency medical care, we are going to use a structure known as a *heap*. A heap is essentially a binary tree that is loosely sorted in either an ascending or descending direction. An example of the index positions of a heap can be seen in the following *Figure 7.10*:

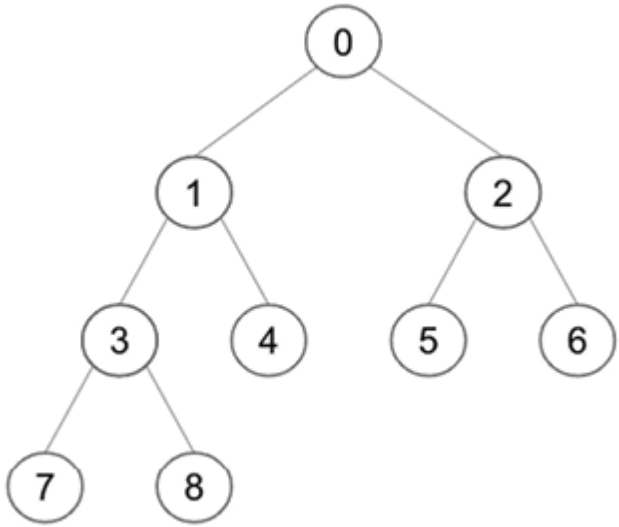

***Figure 7.10**: A visual representation of a 9-item heap's index positions on a binary tree*

The indices of a heap are arranged such that a node's children can be easily computed. The index of the left child is equal to the current index x 2, + 1. Likewise, the index of the right child is equal to the current index x 2, + 2. For example, consider the indexed node 3 in *Figure 7.10*. The left child's index is (3 x 2) + 1, which equals 7. The right child's index is (3 x 2) + 2, which equals 8.

There are two types of heaps: a minimum *min* heap and a maximum *max* heap. The idea with a min heap is that each node should be less than its children, while a max heap's nodes should always be greater than their children. Thus, a min heap has the lowest value at its root node (index of 0), and a max heap has the largest value at its root node.

Note: As we are going to use the min heap approach for this problem, it should be assumed going forward that all mentions of a heap and its functions are in reference to a min heap.

When a new value is added to the heap, it is added at the bottom, and a *heapify* process is triggered. The heapify process starts at the greatest heap index of a non-leaf node, and checks to make sure

that the value at that node is less than both of its children. If it is not, the lower value child is swapped with the value at the current index. The logic behind the heapify algorithm is depicted in *Figure 7.6*.

Writing a Heap class to handle this structure and logic would be complex, time-consuming, and would violate the *small problem* approach of this book. Fortunately, Java comes with a **PriorityQueue** library that can solve a lot of this for us.

Before we get that far, we need to define our **Patient** class. This is the class that we will use to keep track of our patient entity data. It should have two class properties: a string named **name**, and an integer named **priority**. We will need getter and setter accessor methods defined for those properties as well. The **Patient** class public constructor should accept parameters for both **name** and **priority**, and should simply set the class properties with the values of those like-named parameters.

Next, we will need to build a comparator to help our **PriorityQueue** object(s) know how to determine which patient has the most urgent (lowest) priority. To this end, we will build a new class named **PatientComparator**. It will import the **java.util.Comparator** library, and implement the **Comparator** interface with a class type **Patient**. The only requirement for our **PatientComparator**, is to override the **compare()** method from the **Comparator** interface. Essentially, the **compare()** method should return an integer and accept two **Patient** parameters named **p1** and **p2**, respectively.

Inside the **compare()** method, we will use an if construct to check the value returned from the **getPriority()** methods of the **p1** and **p2** objects. If **p1**'s priority is > **p2**'s priority, we will return a value of 1. If **p1**'s priority is < **p2**'s priority, we will return a value of -1. Otherwise (else) we can assume **p1**'s priority is equal to **p2**'s priority, and we will return a value of 0. This will allow our **Patient** objects to be adequately compared within the **PriorityQueue** class.

Now, we can finally build our **MedicalHeapJava** class. It should import the **java.util.PriorityQueue** library, and define a **PriorityQueue** class property of type **Patient**, named **patientHeap**. It should have a constructor that accepts an array of **Patient** named **patientArray**. Inside, the constructor should instantiate the

patientHeap property with a new **PriorityQueue**, while passing a new **PatientComparator** object as a parameter. We will then use a for-each loop to iterate through all of the **Patients** in the **patientArray** to add each one to the **patientHeap**.

Similar to our **Stack** class (as discussed in *Chapter 5, Basic Data Structures*), we should also define methods to push (add) an item to the heap and pop (remove) an item from the heap. Therefore, we will build a new method named **push**, which will accept a **Patient** parameter named **patient**. It will simply invoke the **patientHeap**'s **add()** method (with **patient** as a parameter) to add **patient** to the **patientHeap**.

Next, we will define a method named **pop**, which does not accept any parameters and should return a **Patient** object. This method simply returns the **Patient** object returned from **patientHeap**'s **poll()** method. As this particular solution is built on a min heap, the **pop()** method should always return the **Patient** with the lowest remaining **priority** on the heap.

With these classes in place, our test code (from the beginning of this section) should now produce the expected results.

Conclusion

In this chapter, we further discussed the subject of data structures, taking a look at more advanced structures. We started out by revisiting the linked list. While we had worked with a one-way linked list earlier, we progressed on to build a two-way linked list, and showed an example of how a web browser might use one to back its browsing history, as well as the functionality behind its forward and back buttons.

Next, we moved on to binary trees. This structure relied on a fair amount of self-referential object usage to establish the basic functionality. Using a binary tree, we were able to build an efficient in-memory storage structure for a fruit/grocery example.

Finally, we progressed to a heap, which is a special kind of binary tree. With the heap, we introduced concepts like min heaps and max heaps, and the heapify process. We also discussed comparators and showed how to leverage their functionality to support the

comparison of custom object types. We managed to build a heap-based example using Java's PriorityQueue library to support an emergency medical treatment queue for patients.

Data structures are present in all aspects of computing. While it is not common to have to build one from scratch, it is quite common for real-world problems to require solutions that are built on fundamental structures like linked lists, trees, and queues. Understanding how those structures work at a fundamental level will not only accelerate how fast we can deliver those solutions, but also how well we can teach and convey that understanding to others.

In the next chapter, we will focus on techniques used to enforce a sort order on our data. In fact, we will even use one of the data structures covered in this chapter and leverage its natural, intrinsic sorting capabilities. In addition to sorting, we will also briefly cover ways to classify algorithms in terms of their space and time efficiency.

Points to remember

Have a look at some key points to remember:

- A two-way linked list is one that can be traversed both forward and backward.
- Self-referential objects in Java are a great way to make entities aware of the other entities that are both behind and in front of them.
- A comparator is a special type of Java class that allows certain data structures the ability to assess the value of and rank entities of custom classes and types.
- A binary tree's node values are always between the values of its children.
- A binary tree is different from a B-Tree (balanced tree, commonly used in database indexing).
- A heap is a special kind of binary tree. It has the shape properties of a binary tree, but the value of a node is always

less than (or equal to) each of its children in a min heap.

- Heapify is a process by which a heap sorts its child nodes.

Questions

Answer the following questions to check your understanding:

1. Considering Program 7.1, assume that the user browses backward a few times and then navigates to a new URL. How could we modify the addHistoryItem() method to handle that?
2. For Program 7.2, if we built a deleteByKey() method for our BinaryTree class, how could we account for orphaned child nodes?
3. Our BinaryTree class does not have a constructor. What might a constructor look like that builds the tree from an array of Nodes?
4. How would we have to change Program 7.3 for its underlying structure to be a max heap?
5. What would a parameterless constructor for the MedicalHeapJava class look like? How could we populate the heap?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Chapter 8

Sorting

Introduction

This chapter will focus on applying different approaches to sorting. We will start with a simple algorithm, and then progress into more advanced and efficient methods for enforcing a sort order.

Before proceeding into this chapter, it is recommended to have read the two chapters that covered data structures (*Chapter 5, Basic Data Structures,* and *Chapter 7, Complex Data Structures*). Some of the structures discussed in those chapters automatically lend themselves toward sorting, and we will use them in this chapter as a foundation for some of the solutions. Therefore, it is important to have that understanding before progressing into this chapter.

Structure

This chapter will cover the following programs:

- Program 8.1: Sort users with a bubble sort
- Program 8.2: Sort users with a merge sort
- Program 8.3: Sort with a binary tree sort

Objectives

The primary goal of this chapter is for the reader to learn about different approaches to sorting data. We will also briefly discuss its *Big O* notation for addressing the space-time efficiency of algorithms. Additionally, we will show how these sorting methods can be used.

Program 8.1: Sort users with a bubble sort

Take an array of user's last names, and sort them according to their name using a bubble sort. We can test it with code similar to the following:

```
String[] users = { "Lowell", "Jacobs", "Scott", "Tripp", "Gartman",
"Francis",
        "Samuels, D.", "Samuels, B.", "Murphy", "Roberts", "Knutson",
"Woods",
        "Banks" };
String[] usersBubbleSorted = bubbleSort(users);
System.out.println("Bubble Sort:");
printStringArray(usersBubbleSorted);
```

This should produce an output similar to the following:

```
Bubble Sort:
Banks
Francis
Gartman
Jacobs
Knutson
Lowell
Murphy
Roberts
Samuels, D.
Samuels, B.
```

Scott

Tripp

Woods

Algorithm

To solve this problem, we will take the following steps:

1. Define a new method named **bubbleSort**. It should both return an array of type string, and accept an array of string named **array** as a parameter.
2. Inside the **bubbleSort()** method, define two variables:
    a. A Boolean named **flippedAPair** with a value of **false**.
    b. An integer named **index** with a value of 0.
3. Build a while loop that iterates so long as the value of **index** is < the value at **array.length - 1**.
    a. Inside the while loop, use an if check with the string **compareTo()** method to compare the lexical value of **array[index]** to **array[index + 1]**. Within the if statement:
        i. If this comparison yields a value > 0, we will flip the position of the two items inside the array.
        ii. Define a new string named **temp**, and set it to **array[index]**.
        iii. Set **array[index]** to **array[index + 1]**.
        iv. Set **array[index + 1]** to **temp**.
        v. Set **flippedPair** to a value of **true**.
    b. At this point inside the while loop, we will increment **index**.
    c. At the end of the loop, use an if check to assess whether **index** has a value that is >= **array.length - 1** and **flippedPair** has a value of **true**.
        i. If these conditions are true, then we need to perform another pass through the array. To ensure this happens, set **index** to a value of 0 and

flippedPair to a value of **false**.

Flowchart

The flowchart for *Program 8.1* can be seen in *Figure 8.1*:

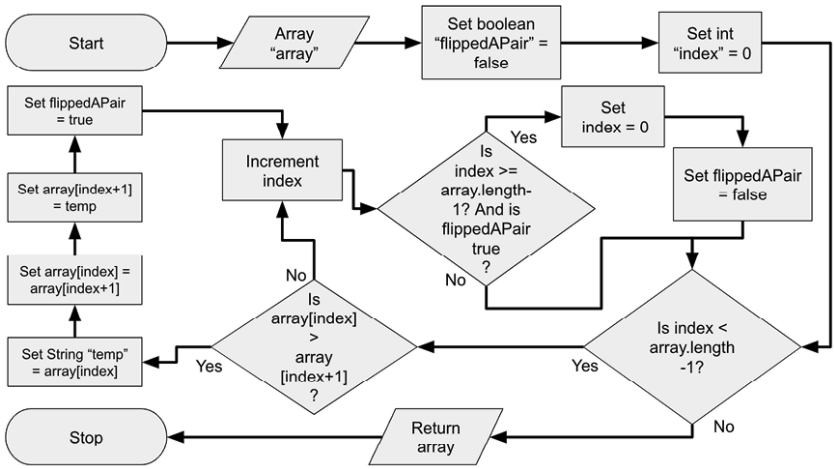

***Figure 8.1**: Flowchart describing the bubbleSort() method*

Java code

The Java code for *Program 8.1*'s **bubbleSort()** and **printStringArray()** methods is specified in the following section:

```
private static String[] bubbleSort(String[] array) {
    boolean flippedAPair = false;
    int index = 0;
    while (index < array.length - 1) {
        if (array[index].compareTo(array[index + 1]) > 0) {
            String temp = array[index];
            array[index] = array[index + 1];
            array[index + 1] = temp;
            flippedAPair = true;
        }
        index++;
```

```
        if (index >= (array.length - 1) && flippedAPair) {
            index = 0;
            flippedAPair = false;
        }
    }
    return array;
}
private static void printStringArray(String[] array) {
    for (String item : array) {
        System.out.println(item);
    }
    System.out.println();
}
```

Code explanation

For sorting our array of users' names, we will use a bubble sort. A bubble sort works on the premise of iterating through each item in the array and comparing them. If the item we are looking at is greater than the next item in the array, the two items are flipped or swapped. Once we reach the end of the array, we restart from its beginning and perform the same checks on each item again. If we can iterate through the entire array without having to flip a pair of adjacent items, then we can consider it *sorted*.

We will start by defining a new method named **bubbleSort**. It should return an array of strings and also accept an array of strings named **array** as a parameter. Inside the method, we will define a Boolean variable named **flippedAPair**, and set it to a value of **false**. Here, we will also define an integer variable named **index**, and set it to a value of 0.

Next, we will build a while loop that will iterate through the array as long as the **index** is < the length of the **array.length - 1**. The idea behind the while condition is that we want to continue checking items in **array** as long as **index** is not at the end of **array**.

Inside the while loop, we will check if the item at **index** is greater

than the next item (**index + 1**). If we were sorting numbers, we could use the greater than sign >. But strings do not consider > to be a valid operator for comparison, so we will have to use the string **compareTo()** method of **array[index]** while passing **array[index +1]** as a parameter. This method compares the strings *lexicographically*, essentially decoding the string into tokens of the ASCII/UTF codes for each character. If **array[index]** is determined to be > **array[index+1]**, then the **compareTo()** method returns the positive difference. Otherwise, it returns the negative difference.

In any case, if the lexicographical value of the item at **array[index]** is > the item at **array[index+1]**, we want to flip the two items. To do this, we first need to initialize a new string named **temp** with the string at **array[index]**, as a temporary place to hold it. Next, we will set **array[index]** to the item at **array[index+1]**. We will then set **array[index+1]** to **temp**. With the *flipping* of the items complete, we can set **flippedAPair** to a value of **true**.

After the if check, we need to increment the **index**. We then need to check if we have reached the last item in **array** and if **flippedAPair** is equal to a value of **true**. If so, this means that we need to go through the items of **array** at least one more time, so we need to reset **index** to 0 and **flippedAPair** to **false**. If we have reached the last item in the **array** without having to flip a pair (**flippedAPair** is **false**), the loop will end, and we can return **array** to the calling method.

We will also need to build a method named **printStringArray** that accepts an array of **String** named **array** as a parameter. Inside the method, we will build a for-each loop that iterates through each array item named **item**, and simply prints it via a call to the **System.out.println()** method. This should now allow us to sort the original array and print its contents to the system console.

As we need to iterate through the array several times, it should be apparent that a bubble sort is not terribly efficient. In fact, its *Big O* notation is $O(n^2)$, meaning that the time complexity required to sort the array is proportional to the number of items squared.

Note: The Big O notation is a common way of discerning the worst-case potential time complexity and space requirements of algorithms. It most commonly shows a capital letter O along with a simple arithmetic operation on the lowercase letter n, where n

represents the number of items to be processed. O(1) is considered to be the most efficient, with notations like O(n2), O(2n), and O(n!) considered to be inefficient.

While not optimal, a bubble sort is still important to understand. Its lack of complexity helps with code readability and also makes it a great starting point for learning concepts around sorting. However, it probably should not be used to sort large numbers of items when time and performance are a factor.

Program 8.2: Sort users with a merge sort

Take an array of users' last names, and sort them according to their name using a merge sort. We can test it with code similar to the following:

```
String[] users = { "Lowell", "Jacobs", "Scott", "Tripp", "Gartman",
"Francis",
        "Samuels, D.", "Samuels, B.", "Murphy", "Roberts", "Knutson",
"Woods",
        "Banks" };
String[] usersMergeSorted = bubbleSort(users);
System.out.println("Merge Sort:");
printStringArray(usersMergeSorted);
```

This should produce an output similar to the following:

```
Merge Sort:
Banks
Francis
Gartman
Jacobs
Knutson
Lowell
Murphy
Roberts
Samuels, D.
```

Samuels, B.

Scott

Tripp

Woods

Algorithm

To solve this problem, we will take the following steps:

1. Define a new method named **splitArray** that returns an array of strings. It should accept the following parameters:
    a. An array of strings named **array**.
    b. An integer named **start**.
    c. An integer named **end**.
    a. Define a new string array named **returnVal**, initialized with the size determined by the result of **end - start**.
    b. Define an integer named **returnIndex** with an initial value of 0.
    c. Build a for loop that:
        i. Defines an integer named **index** set with the value of **start**.
        ii. Iterates as long as **index** is < **end**.
        iii. Increments **index**.
    d. Inside the loop, set **returnVal[returnIndex]** equal to the value of **array[index]**, and then increment **returnIndex**.
    e. Return **returnVal** to the calling method.
2. Define a new method named **mergeSort** that returns an array of strings and accepts an array of strings named **array** as a parameter.
    a. Write an if statement to check if **array.length** is > 1. If true:
        i. Define a new integer named **middle** as the result of **array.length / 2**.
        ii. Define a new array of strings named leftArray, set

to the result of a call to the **splitArray()** method, while passing the parameters of the **array**, **0**, and **middle**.

iii. Define a new array of strings named **rightArray**, set to the result of a call to the **splitArray()** method, while passing the parameters of **array**, **middle**, and **array.length**.

iv. Now, set **leftArray** to the result of a recursive call to **mergeSort()** with a parameter of **leftArray**.

v. Likewise, set **rightArray** to the result of a recursive call to **mergeSort()** with a parameter of **rightArray**.

vi. Next, initialize three integer variables named **leftIndex**, **rightIndex**, and **mergedIndex**, setting them all to 0.

b. Next, define a while loop that:

i. Iterates as long as both conditions are true, **leftIndex** < **leftArray.length** and **rightIndex** < **rightArray.length**.

c. Next, write an if statement using the string **compareTo()** method to check if the difference between **leftArray[leftIndex]** and **rightArray[rightIndex]** is < 0.

i. If the preceding is **true**, set **array[mergedIndex]** to **leftArray[leftIndex]**. Otherwise, set **array[mergedIndex]** to **rightArray[rightIndex]**. Then, increment **mergedIndex**.

d. Next, define a for loop that:

i. Defines an integer named **index** set with the value of **leftIndex**.

ii. Iterates as long as **index** is < **leftArray.length**.

iii. Increments **index**.

iv. Inside the loop, set **array[mergedIndex]** to **leftArray[index]**. Then, increment **mergedIndex**.

e. Then, define a for loop that:

i. Defines an integer named **index** set with the value of **rightIndex**.

ii. Iterates as long as **index** is < **rightArray.length**.

iii. Increments **index**.

iv. Inside the loop, set **array[mergedIndex]** to **rightArray[index]**. Then, increment **mergedIndex**.

f. Return **array** to the calling method.

Flowchart

The flowchart for *Program 8.2* can be seen in *Figures 8.2*, *8.3.a*, and *8.3.b*:

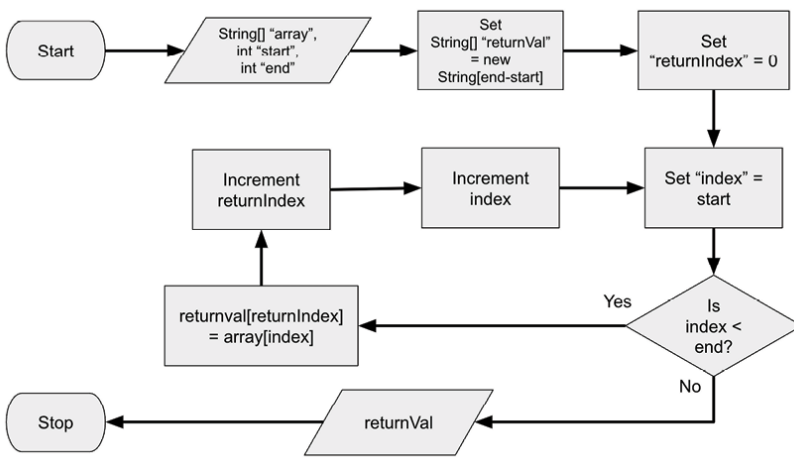

***Figure 8.2**: Flowchart describing the splitArray() method*

Flowcharts describing the first half as well as the second half of the **mergeSort()** method:

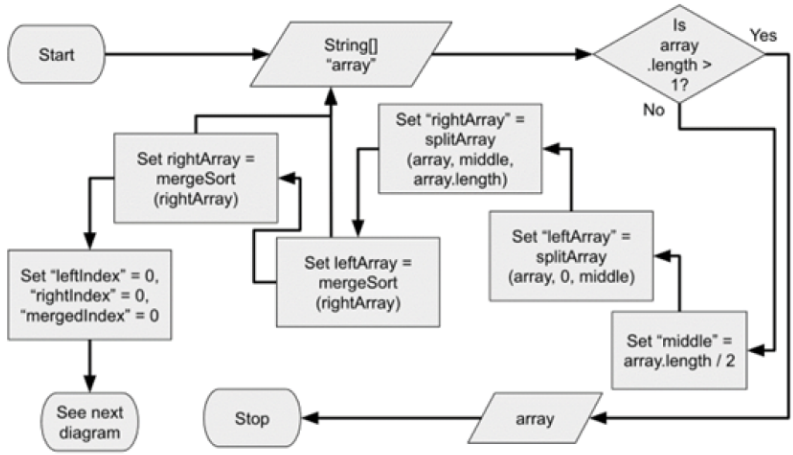

***Figure 8.3.a**: Flowchart describing the first half of the mergeSort() method*

The following figure highlights the core gameplay loop, illustrating how threat management and player decisions drive sustained survival:

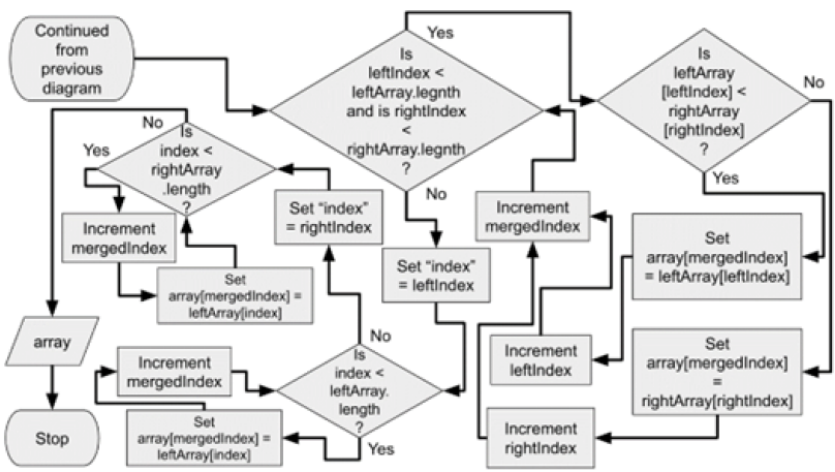

***Figure 8.3.b**: Flowchart describing the second half of the mergeSort() method*

Java code

The Java code for *Program 8.2*'s **splitArray()** and **mergeSort()** methods is specified in the following section:

```
private static String[] splitArray(String[] array, int start, int end) {
```

```
    String[] returnVal = new String[end - start];
    int returnIndex = 0;
    for (int index = start; index < end; index++) {
        returnVal[returnIndex] = array[index];
        returnIndex++;
    }
    return returnVal;
}
private static String[] mergeSort(String[] array) {
    if (array.length > 1) {
        int middle = (array.length / 2);
        String[] leftArray = splitArray(array, 0, middle);
        String[] rightArray = splitArray(array, middle, array.length);
        leftArray = mergeSort(leftArray);
        rightArray = mergeSort(rightArray);
        int leftIndex = 0;
        int rightIndex = 0;
        int mergedIndex = 0;
        while (leftIndex < leftArray.length && rightIndex <
rightArray.length) {
            if (leftArray[leftIndex].compareTo(rightArray[rightIndex])
< 0) {
                array[mergedIndex] = leftArray[leftIndex];
                leftIndex++;
            } else {
                array[mergedIndex] = rightArray[rightIndex];
                rightIndex++;
            }
            mergedIndex++;
        }
```

```
      for (int index = leftIndex; index < leftArray.length; index++)
{
         array[mergedIndex] = leftArray[index];
         mergedIndex++;
      }
      for (int index = rightIndex; index < rightArray.length; index++) {
         array[mergedIndex] = rightArray[index];
         mergedIndex++;
      }
   }
   return array;
}
```

Code explanation

For sorting our array of users' names this time, we will use a merge sort. A merge sort works by splitting the array of items recursively in half. For instance, a 16-item array will be split into two 8-item arrays, then into four 4-item arrays, and finally into eight 2-item arrays. A 2-item array is the smallest size for the sub-arrays.

Pairs of the small arrays are then matched up side-by-side, compared, and then merged back together in order from smallest to largest. As this happens, the arrays become larger and larger, until all items are merged back into the original array. When the merge is complete, the array is considered to be *sorted*.

For the code, we will first need to write a method to split an array into smaller parts. We will write a new method in our main program named **splitArray**. This new method will return an array of strings, and accept an array of strings named **array**, and two integers named **start** and **end** as parameters. The idea here is that we will take the starting and ending indexes with an array, and return the items from **start** to **end - 1** in a new array.

The size of the new string array named **returnVal** can be computed as **end - start**. We will also need an integer to store the current value of the index for **returnVal**, named **returnIndex**. The primary

part of the **splitArray()** method will be a for loop, which will iterate an integer named **index** from **start** to **end**. Inside the loop, we will essentially copy the items indexed at start through end to our **returnVal** array. When complete, we will return **returnVal** to the calling method.

Next, we will build a new method named **mergeSort**, which returns an array of strings and accepts an array of strings named **array** as a parameter. The primary part of the **mergeSort()** method is an if statement that ensures the **array** has a size greater than 1. Inside the if statement, we first want to compute the *middle* of the array indexes by initializing a new integer variable named **middle** with the result of **array.length** divided by 2.

Then, we will instantiate two new string arrays named **leftArray** and **rightArray**, setting them to the results of the **splitArray()** method. The **leftArray** will be built with array items **0** through **middle - 1**, and **rightArray** with items **middle** through **array.length - 1**. With the array split, we will then set both **leftArray** and **rightArray** to the results of respective recursive calls to the **mergeSort()** method with **leftArray** and **rightArray** as parameters.

We will then begin to merge **leftArray** and **rightArray** back together into the original **array** variable/parameter. First, we will instantiate three integer variables to 0, named **leftIndex**, **rightIndex**, and **mergedIndex**.

Next, we will build a while loop that iterates on the conditions of both **leftIndex** < **leftArray.length** and **rightIndex** < **rightArray.length**. Inside the loop, we will have an if statement that uses the string **compareTo()** method to see if the lexicographical difference of **leftArray[leftIndex]** and **rightArray[rightIndex]** is < 0. If it is, we will set **array[mergedIndex]** to **leftArray[leftIndex]**, and then increment **leftIndex**. Otherwise, we will set **array[mergedIndex]** to **rightArray[rightIndex]**, and then increment **rightIndex**. Outside of the if statement, we will increment **mergedIndex**.

Once we have made it through the while loop, we are likely to have some left-over items that still need to be merged back into the array. We can accomplish this with two simple for loops that start at the current values of **leftIndex** or **rightIndex**, and iterate to the end of

their respective arrays (**leftArray** or **rightArray**). At this point, there should be remaining items only in **leftArray** or only in **rightArray**, but not both. Therefore, it does not really matter which for loop (left or right) runs first.

After those for loops have run, all items should be returned to **array**, in an ascending, sorted order. With the if statement's logic now complete, we can return **array** to the calling method or program. Running the test code presented at the beginning of this section should now produce the desired results.

The time complexity for a merge sort (in *Big O* notation) in the worst case is *O(n log n)*. This means that the time for the items to be sorted grows logarithmically, relative to the number of items to be sorted (*n*), multiplied by the number of items (*n*). This makes it faster and more efficient than the bubble sort.

Program 8.3: Sort with a binary tree sort

Using a binary tree sort, sort players from a hockey team (by their last name) by adding them to a binary tree and then printing the sorted result. We can test it with code similar to the following:

```
BinaryTree tree = new BinaryTree();
tree.insert(new Player(66, "Lemieux", "Mario", "C"));
tree.insert(new Player(25, "Stevens", "Kevin", "LW"));
tree.insert(new Player(68, "Jagr", "Jaromir", "RW"));
tree.insert(new Player(55, "Murphy", "Larry", "D"));
tree.insert(new Player(5, "Samuelsson", "Ulf", "D"));
tree.insert(new Player(30, "Barrasso", "Tom", "G"));
tree.insert(new Player(10, "Francis", "Ron", "C"));
tree.insert(new Player(7, "Mullen", "Joe", "RW"));
tree.insert(new Player(15, "McEachern", "Shawn", "LW"));
List<Player> playerList = tree.toList();
printPlayerList(playerList);
```

This should produce an output similar to the following:

30 Tom Barrasso - G

10 Ron Francis - C

68 Jaromir Jagr - RW

66 Mario Lemieux - C

15 Shawn McEachern - LW

7 Joe Mullen - RW

55 Larry Murphy - D

5 Ulf Samuelsson - D

25 Kevin Stevens – LW

Algorithm

To solve this problem, we will take the following steps:

1. Define a new class named **Player**. This class should have:
    a. The following class properties:
        i. An integer named **jersey**.
        ii. A string named **lastName**.
        iii. A string named **firstName**.
        iv. A string named **position**.
        v. A self-referential object of class **Player** named **leftNode**.
        vi. A self-referential object of class **Player** named **rightNode**.
    b. A public constructor that accepts the following parameters:
        i. An integer named **jersey**.
        ii. A string named **lName**.
        iii. A string named **rName**.
        iv. A string named **pos**.
        v. Sets the class properties to the like-named parameters.
    c. Public getter accessor methods for all class properties.

d. Public setter accessor methods for the **jersey**, **lastName**, **firstName**, and **position** class properties.

e. A public method named **getFullName,** which returns the string result of a concatenation of the **lastName** and **firstName** class properties.

f. A public method named **insert** that accepts an object of class **Player** named **newNode**. This method will have a single if-statement, with one else-if condition and one else condition.

g. If the result of the string **compareTo()** of **newNode**'s **getFullName()** method is < the result of the current **Player**'s **getFullName()** method:

   i. If the **leftNode** property is null, set **leftNode** to **newNode**. Otherwise, call **leftNode**'s **insert()** method while passing **newNode** as a parameter.

h. Else, if the result of the string **compareTo()** of **newNode**'s **getFullName()** method is | the result of the current **Player**'s **getFullName()** method:

   i. If the **rightNode** property is null, set **rightNode** to **newNode**. Otherwise, call **rightNode**'s **insert()** method while passing **newNode** as a parameter.

   ii. Otherwise, the result of **newNode**'s **getFullName()** method must be equal to the result of the current **Player**'s **getFullName()** method. In this case, simply set the current **Player**'s **jersey** and **position** properties to the result of **newNode**'s **getJersey()** and **getPosition()** methods, respectively.

i. A public method named **toString()** that returns a string. The method will override the default **toString()** method. Inside this method:

   i. Define a new **StringBuilder** named **returnVal**.

   ii. Using the **StringBuilder**'s **append()** method, append the **jersey**, **firstName**, and **lastName** class properties, separated by a space.

iii. Then, use the **StringBuilder**'s **append()** method to append a dash character and then the **position** class property.

2. Define a new class named **BinaryTree**. It should have the following:

   a. Imports for **java.util.ArrayList** and **java.util.List**

   b. A class property of an object of the **Player** class named **root**.

   c. A public method named insert that accepts a **Player** object named **newNode** as a parameter. This method should have a single if statement:

      i. If the **root** class property is null, set **root** to **newNode**.

      ii. Else, call **root**'s **insert()** method while passing **newNode** as a parameter.

   d. A public method named **toList** that returns a **List** object of type **Player**. This method simply returns the result of the **getNodesAsList()** method while passing **root** as a parameter.

   e. A private method named **getNodesAsList,** which returns a **List** object of type **Player**, and accepts an object of the **Player** class named **node**.

   f. Inside the previous method, define a new **List** of type **Player** named **tempList** as a new **ArrayList**. If **node** is not null:

      i. Call **tempList**'s **addAll()** method while passing a parameter of the result of a recursive call to **getNodesAsList()** with the result of **node**'s **getLeftNode()** method as a parameter.

      ii. Call **tempList**'s **add()** method while passing **node** as a parameter.

      iii. Call **tempList**'s **addAll()** method while passing a parameter of the result of a recursive call to **getNodesAsList()** with the result of **node**'s **getRightNode()** method as a parameter.

iv. Return **tempList** to the calling method.

3. In our main program, write a new method named **printPlayerList** that accepts a **List** of type **Player** named **list** as a parameter.

   a. Be sure to import the **java.util.List** library.

   b. Using a for-each loop, iterate through each **Player** named **player** from **list**:

   c. Using the **System.out.println()** method, print **player** to the console.

   d. Using the **System.out.println()** method, print a line break.

Flowchart

The flowcharts for *Program 8.3* can be seen in *Figures 8.4* and *8.5*:

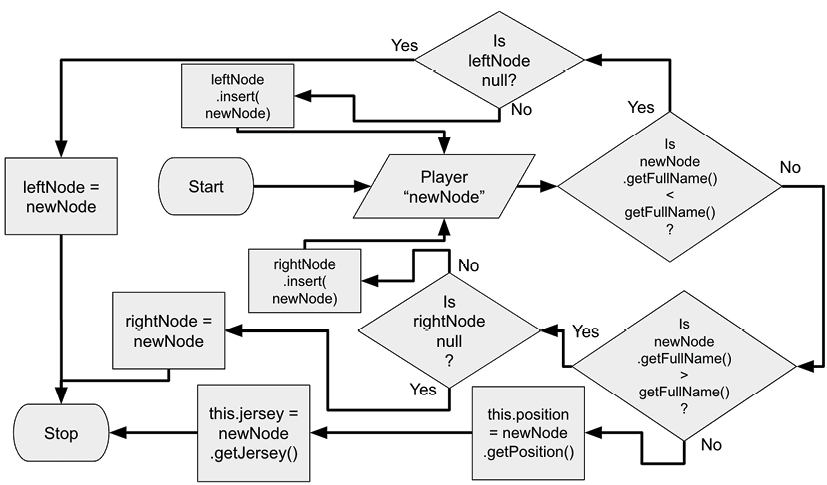

***Figure 8.4**: Flowchart describing the insert() method from the Player class*

The following figure shows the logic for traversing player nodes and building a temporary list:

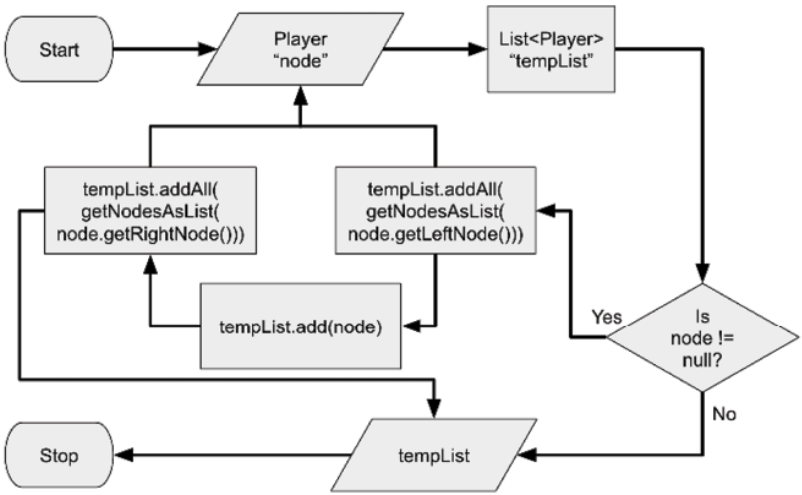

***Figure 8.5**: Flowchart describing the getNodesAsList() method from the BinaryTree class*

Notice how the **tempListAddAll()** calls redirect back to the beginning, indicating the recursive call.

Java code

The Java code for *Program 8.3*'s **Player** class can be seen as follows:

```
public class Player {
    private int jersey;
    private String lastName;
    private String firstName;
    private String position;
    private Player leftNode;
    private Player rightNode;
    public Player(int jersey, String lName, String fName, String pos) {
        this.jersey = jersey;
        this.lastName = lName;
        this.firstName = fName;
        this.position = pos;
```

```
    }
    public void insert(Player newNode) {
        if ((newNode.getFullName()).compareTo(getFullName()) < 0) {
            if (leftNode == null) {
                leftNode = newNode;
            } else {
                leftNode.insert(newNode);
            }
        } else if ((newNode.getFullName()).compareTo(getFullName())
> 0) {
            if (rightNode == null) {
                rightNode = newNode;
            } else {
                rightNode.insert(newNode);
            }
        } else {
            this.jersey = newNode.getJersey();
            this.position = newNode.getPosition();
        }
    }
    public String getFullName() {
        return lastName + firstName;
    }
    public int getJersey() {
        return jersey;
    }
    public void setJersey(int jersey) {
        this.jersey = jersey;
    }
    public String getLastName() {
```

```
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getPosition() {
        return position;
    }
    public void setPosition(String position) {
        this.position = position;
    }
    public Player getLeftNode() {
        return this.leftNode;
    }
    public Player getRightNode() {
        return this.rightNode;
    }
}
```

The Java code for the **BinaryTree** class is shown as follows:

```
import java.util.ArrayList;
import java.util.List;
public class BinaryTree {
    private Player root;
    private List<Player> playerList;
```

```
    public void insert(Player newNode) {
        if (root == null) {
            root = newNode;
        } else {
            root.insert(newNode);
        }
    }
    public List<Player> toList() {
        if (playerList == null) {
            playerList = new ArrayList<>();
        } else {
            playerList.clear();
        }
        return getNodesAsList(root);
    }
    private List<Player> getNodesAsList(Player node) {
        List<Player> tempList = new ArrayList<>();
        if (node != null) {
            tempList.addAll(getNodesAsList(node.getLeftNode()));
            tempList.add(node);
            tempList.addAll(getNodesAsList(node.getRightNode()));
        }
        return tempList;
    }
}
```

The Java code for the **printPlayerList()** method can be seen as follows:

```
private static void printPlayerList(List<Player> list) {
    for (Player player : list) {
        System.out.println(player);
```

```
    }
    System.out.println();
}
```

Code explanation

To solve this problem, we will need to build two classes: a class named **Player** and a class named **BinaryTree**. The **Player** class should have class properties defined for an integer named **jersey**, a string named **lastName**, a string named **firstName**, and a string named **position**. It should also have two self-referential property objects of type **Player** named **leftNode** and **rightNode**.

The class properties will also need accessor methods. We will create getter and setter accessor methods for **jersey**, **lastName**, **firstName**, and **position**. However, the **leftNode** and **rightNode** properties should only need getters.

The **Player** class should have a public constructor that accepts an integer parameter named **jersey**, a string parameter named **lName**, a string parameter named **fName**, and a string parameter named **pos**. The constructor should use these parameters to define the similarly named class properties.

This class should have a public method named **getFullName** that returns a string. All the method needs to do is return the string concatenation result of the **lastName** and **firstName** (in that order) class properties.

The **Player** class should also have a public method named **insert** that accepts a **Player** object named **newNode** as a parameter. This method should use an if check to assess whether or not the value returned by **newNode**'s **getFullName()** method is < the result of the (local) **getFullName()** method. If that is true, another nested if statement should be used to see if the **leftNode** property is null. If it is, **leftNode** should then be set to **newNode**. If **leftNode** has a value (is not null), then we should call **leftNode**'s **insert()** method while passing **newNode** as a parameter.

The next part of the if statement should assess whether or not the value returned by **newNode**'s **getFullName()** method is > the result returned by the (local) **getFullName()** method. If that is true,

another, nested if statement should be used to see if the **rightNode** property is null. If it is, **rightNode** should then be set to **newNode**. If **rightNode** has a value (is not null), then we should call **rightNode**'s **insert()** method while passing **newNode** as a parameter.

Finally, we should have an else construct which assumes that the value returned by **newNode**'s **getFullName()** method is equal to the result of the (local) **getFullName()** method. In that case, we should assume we are overwriting all data stored at this node and set the **jersey** and **position** class properties to the results from **newNode**'s **getJersey()** and **getPosition()** methods, respectively.

Note: For our binary tree, the data is being sorted at insert-time.

The final required public method for the **Player** class is named **toString**, which will return a string. This method will override the default **toString()** method that all Java objects inherit (from the base **Object** class). Inside this method, we will define a new **StringBuilder** named **returnVal**. The idea here is to append the following:

- **jersey** | A space
- **firstName** | A space
- **lastName** | A space, a dash, and a space
- **position**

Next, we will define a new class named **BinaryTree**. This class will be very similar to the **BinaryTree** class from *Chapter 7, Complex Data Structures*. It will have a single class property of the **Player** class named **root**.

Inside the **BinaryTree** class, we will need a public method named **insert** that accepts a parameter of the **Player** class named **newNode**. This method simply checks to see if **root** is equal to null, and if true, sets **root** equal to **newNode**. If **root** has a value, then we will call its **insert()** method while passing **newNode** as a parameter.

Next, we will define a private method named **getNodesAsList** that returns a **List** of type **Player**, and accepts a **Player** object named **node** as a parameter. Inside this method, we will define a new **List** of type **Player** named **tempList** as a new **ArrayList**. If **node** is not null, we will perform the following three tasks:

- Recursively call the **getNodesAsList()** method while passing the result of node's **getLeftNode()** method. We will then use **tempList**'s **addAll()** method to add the result to **tempList**.
- Add **node** to **tempList**.
- Recursively call the **getNodesAsList()** method while passing the result of **node**'s **getRightNode()** method. We will then use **tempList**'s **addAll()** method to add the result to **tempList**.

After the if statement, we will return **tempList** to the calling method. Calling **getNodesAsList()** recursively in this fashion will allow us to quickly traverse the entire tree and return it as a **List** of **Player**.

The last public method that we will need for the **BinaryTree** class is the **toList** method. This method does not accept any parameters and returns a **List** of **Player**. All it does, is begin the node traversal process by calling the **getNodesAsList()** method with root as a parameter, and returns the result to the calling program.

Finally, in our main program we should write a method named **printPlayerList** that accepts a **List** of **Player** named **list** as a parameter. This method is fairly simple, as it just needs a for each loop to iterate over each **Player** named **player** in **list**, and print **player** to the system console.

Running the test code presented at the beginning of this section should now produce the desired results.

With a Big O notation of *O(log n)*, our binary tree sort operation is significantly faster than our bubble sort. With a binary tree sort, half of the remaining items are eliminated on each iteration. Thus, the possible final location for a new node is significantly reduced as it progresses through each of the different levels of the tree.

Note: Binary trees tend to behave more predictably if the initial root node is close to the median value of the data set. If the root node has a value that is on the extremely high or low end of the data set, the tree becomes unbalanced. This can cause its Big O efficiency to (negatively) rise to O(n).

Conclusion

In this chapter, we examined three sorting algorithms: the bubble sort, the merge sort, and the binary tree sort. We were also able to see each of the approaches perform sorting in action, working with small datasets, making it easy for us to see what is happening in each step.

We also lightly touched on how time and space complexity are measured for algorithms, with the Big O notation. This notation is common throughout our industry, and can be seen in manuals, documentation, and even in Stack Overflow answers.

It is important to remember that Java (and most other third-generation programming languages) have libraries and tools available that mostly negate the need to build sorting methods from scratch, but regardless of the availability of powerful alternatives, understanding what happens underneath a delivered sort routine is extremely important for building high-performing and stable software applications. Plus, sorting problems make great technical interviewing questions.

In the next (and final) chapter, we will focus on solving problems that arise from displaying graphics in Java. We will examine some of the approaches to be used while drawing and animating. We will also learn how to leverage virtual threads to solve some of those problems.

Points to remember

Have a look at some key points to remember:

- The Big O notation is a common way of discerning the worst-case potential time complexity and space requirements of algorithms.
- While easy to understand, a bubble sort is not very efficient and should not be used on large data sets.
- A merge sort works by splitting the primary array into smaller, subarrays, sorting the subarrays, and then merging them back together, recursively.
- A binary tree sort will perform more efficiently if the root node's value is as close to the median value of the entire

data set as possible.

Questions

Answer the following questions to check your understanding:

1. How does the string compareTo() method work?
2. With the bubble sort, why is it important for us to keep track of whether or not we have flipped a pair?
3. Instead of the splitArray() method, is there another, easier way using which we could split an array into subarrays?
4. With the merge sort, we will likely have leftover items after the merge that need to be added to the end of the array. Should we merge the leftover items from leftArray or rightArray first? Why?
5. Which method from the List class should we use to add the items from another List?
6. Write a constructor for the BinaryTree class that accepts a List of Player as a parameter and inserts them.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

CHAPTER 9

Graphics

Introduction

In this last chapter, we will be focusing on using different approaches to solve graphics problems. Graphics and animation can pose some challenging obstacles, as the math behind them can be significant. However, even building simple animations can be fun, making this chapter a great final chapter for the book.

We will utilize the delivered Java Swing library for our graphics and animation. Java Swing has been around for quite a while. While there are other newer alternatives out there for building graphics in Java, Swing is projected to remain a part of the JDK for the foreseeable future. Also, there is plenty of Swing code running in enterprises today.

Note: While JavaFX has made some great strides in producing a graphics library with more up-to-date capabilities, getting it set up as a part of a development environment can be a complex task. Using the standard, delivered Java Swing library will be significantly easier.

Structure

This chapter will cover the following programs:

- Program 9.1: Draw the moon with a starfield
- Program 9.2: Animate a bouncing ball
- Program 9.3: Animate multiple bouncing balls

Objectives

The primary goal of this chapter will be to apply an algorithmic approach to solve problems requiring graphics and animation. Lessons learned in this chapter can be applied to countless problems in the world of data processing. Building this understanding and approach will be one of our two primary goals for this chapter. The other lesson is to have fun!

Program 9.1: Draw the moon with a starfield

Draw a crescent moon with a randomly generated field of stars. We can test our solution with code such as the following:

```
JFrame starfieldFrame = new JFrame();
starfieldFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
StarfieldPanel starfieldPanel = new StarfieldPanel(1500);
starfieldFrame.add(starfieldPanel);
starfieldFrame.pack();
starfieldFrame.setVisible(true);
```

This should produce an output similar to what is shown in *Figure 9.1*:

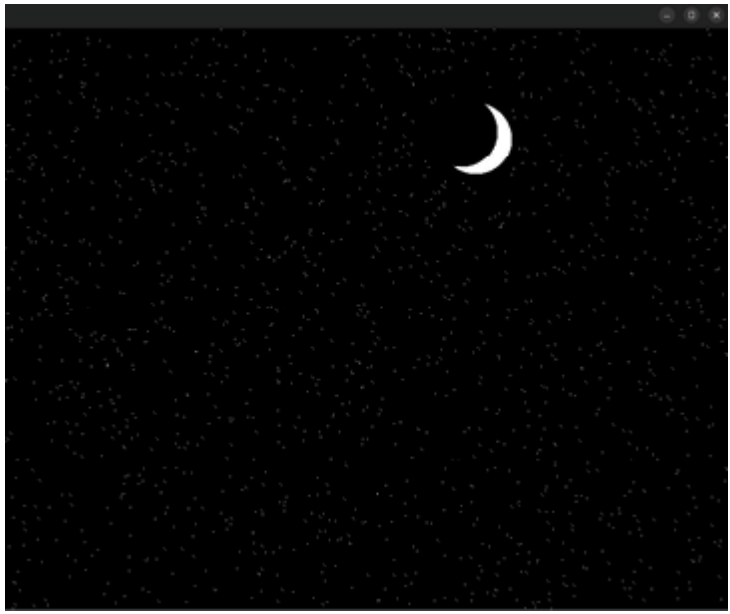

***Figure 9.1**: A window or panel showing a field of stars and a crescent moon*

Algorithm

To solve this problem, we will take the following steps:

1. Define a new class named **StarfieldPanel** that extends the **JPanel** class. This class should have:
    a. Imports for:
        i. **java.awt** (**Color**, **Dimension**, **Graphics**, **Graphics2D**)
        ii. **java.util.Random**
        iii. **javax.swing.JPanel**
    b. The following constants are defined:
        i. A **serialVersionUID** set to a valid long value.
        ii. An integer named **WIDTH** set to a value of **1000**.
        iii. An integer named **HEIGHT** set to a value of **800**.
    c. A single class property; an integer named **stars**. No getter or setter accessor methods are necessary.
    d. A public constructor that accepts an integer named **stars** as a parameter. Inside the constructor:

i. Sets the **stars** class property to the value of the **stars** parameter.

ii. Invokes the **setPreferredSize()** method with a new **Dimension** object, while passing **WIDTH** and **HEIGHT** as parameters, respectively.

iii. Invokes the **setBackground()** method while passing **Color.BLACK** as a parameter.

iv. Invokes the **setFocusable()** method while passing true as a parameter.

e. A method named **paintComponent** that accepts a **Graphics** object named *g* as a parameter. Inside this method:

i. Invoke the superclass's **paintComponent()** method by passing g as a parameter.

ii. Define a **Graphics2D** class object named **g2**, instantiated with the result of casting **g** to be a **Graphics2D** object.

iii. Call the **setColor()** method on the **g2** object while passing **Color.WHITE** as a parameter.

iv. Instantiate a new **Random** class object named **randomInts**.

f. Next, inside the preceding method, define a for loop to set an integer named **starCounter** to **0**, and iterate as long as **starCounter** is < **stars**. Inside the for loop:

i. Define a new integer named **x**, and set it to the result of the **randomInts** object's **nextInt()** method while passing **0** and **WIDTH** as respective parameters.

ii. Define a new integer named **y**, and set it to the result of the **randomInts** object's **nextInt()** method while passing **0** and **HEIGHT** as respective parameters.

iii. Use the **g2** object's **drawLine()** method to draw a line from (**x,y**) to (**x+1,y+1**).

iv. Outside the for loop, use the **g2** object's **fillOval()**

method to draw a circle by passing **600**, **100**, **100**, and **100** as respective parameters.

v. Call the **setColor()** method on the **g2** object while passing **Color.BLACK** as a parameter.

vi. Use the **g2** object's **fillOval()** method to draw a circle by passing **580**, **90**, **100**, and **100** as respective parameters.

vii. Call the **dispose()** method on the **g2** object.

Flowchart

The flowchart for the **StarfieldPanel**'s **paintComponent()** method of *Program 9.1* can be seen in *Figure 9.2*:

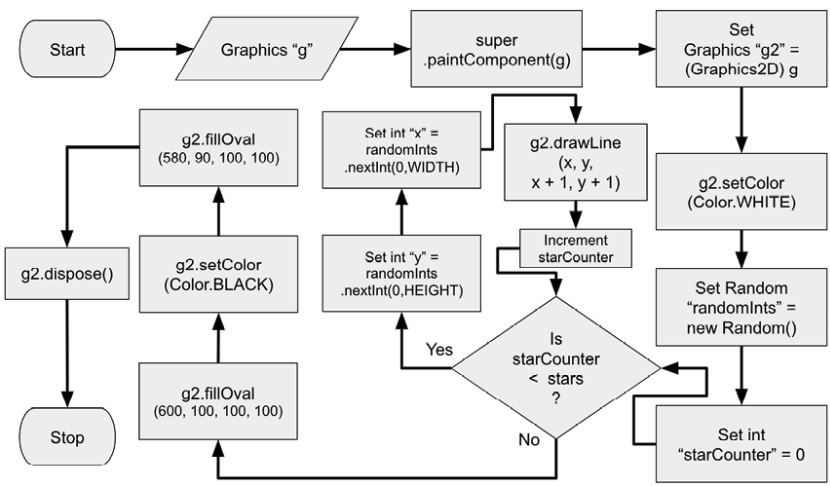

***Figure 9.2**: Flowchart describing the paintComponent() method of the StarfieldPanel class*

Java code

The Java code for *Program 9.1's* **StarfieldPanel** class is specified in the following section:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
```

```
import java.util.Random;
import javax.swing.JPanel;
public class StarfieldPanel extends JPanel {
   private static final long serialVersionUID =
6361841055806902341L;
   private static final int WIDTH = 1000;
   private static final int HEIGHT = 800;
   private int stars;
   public StarfieldPanel(int stars) {
      this.stars = stars;
      this.setPreferredSize(new Dimension(WIDTH, HEIGHT));
      this.setBackground(Color.black);
      this.setFocusable(true);
   }
   public void paintComponent(Graphics g) {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D)g;
      g2.setColor(Color.WHITE);
      Random randomInts = new Random();
      for (int starCounter = 0; starCounter < stars; starCounter++)
{
         int x = randomInts.nextInt(0,WIDTH);
         int y = randomInts.nextInt(0,HEIGHT);
         g2.drawLine(x, y, x+1, y+1);
      }
      g2.fillOval(600, 100, 100, 100);
      g2.setColor(Color.BLACK);
      g2.fillOval(580, 90, 100, 100);
      g2.dispose();
   }
```

```
}
```

Code explanation

To solve this problem, we will first need to define a new class named **StarfieldPanel** that extends the **JPanel** Swing class. As this is a serializable class (meaning that Java can convert it into a byte stream), we will need to define a long constant named **serialVersionUID**. It is recommended to auto-generate the value of **serialVersionUID**. Next, we should define additional integer constants named **WIDTH** and **HEIGHT**, with values of **1000** and **800**, respectively. Additionally, we will need a single class property, an integer named **stars**, to hold the value representing the number of stars that we should draw.

The **StarfieldPanel** class should have a public constructor that accepts an integer named **stars** as a parameter. The **stars** class property should be set to the value of the **stars** parameter. Next, the constructor should define the size of the panel by calling the **setPreferredSize()** method while passing a new **Dimension** object constructed with the parameters of **WIDTH** and **HEIGHT**, respectively. It should then set the background color of the panel by calling the **setBackground()** method while passing **Color.BLACK** as a parameter. Finally, the constructor should call the **setFocusable()** method (to allow the panel to accept input and window focus) while passing a value of **true** as a parameter.

With the **JPanel** all set up, we can now move on to drawing the field of stars. We will implement a public method named **paintComponent** to handle this, and it should accept a **Graphics** object named **g**. First of all, we are going to call the **paintComponent()** method of our super class (**JPanel**). The super class's **paintComponent()** method will take care of painting the entire panel surface, even the spots that we do not paint. If we fail to call it, we will get some splotchy, unpredictable behavior. Usually, this is a good first step anytime you override the **paintComponent()** method.

Note: If we were building a graphics application that did paint the entire panel, we might not need to call the super class's paintComponent() method.

Next, we will define a new object named **g2** from the **Graphics2D** class, instantiating it with the result of casting the standard **Graphics g** object to be a **Graphics2D** object. This is going to give us some extra tools to work with. For our first drawing-related task, we are going to set the current drawing color to **Color.WHITE**. We will also need to instantiate a new **Random** object named **randomInts**, to help us build the star field.

The primary part of this method will be the for loop that we will use to draw the individual stars. We will use it to iterate from **0** to the value of **stars**. Inside, we will define integer variables named **x** and **y** to randomly generate integers to serve as our drawing coordinates. Finally, we will use **g2**'s **drawLine()** method to draw a *star* from the (**x,y**) coordinates to the (**x+1,y+1**) coordinates.

Once we are done with the loop, we can draw the moon. To do this, we will use **g2**'s **fillOval()** method, while passing coordinates of (**600**,**100**) and a width and height of **100**. Since we are drawing an oval with equal height and width, it will appear as a circle. Next, to make it a crescent moon, we will change our drawing color to **Color.BLACK** (matching our background) and draw a same-size oval at the slightly offset coordinates of (**580**,**90**).

Our drawing tasks are now complete. The last thing to do is to call **g2**'s **dispose()** method.

Note: While there is some debate on whether or not calling g2.dispose() is entirely necessary, the prevailing wisdom is to properly dispose of any Graphics/Graphics2D objects that we create ourselves.

With our **StarfieldPanel** class complete, our test code should now produce the output shown in the preceding *Figure 9.1*.

Program 9.2: Animate a bouncing ball

Draw a ball on a panel, start its movement in a random direction, and have it *bounce* off the edges of the panel. We can test it with code similar to the following:

```
JFrame ballFrame = new JFrame();
ballFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
BallPanel ballPanel = new BallPanel(1000,800);
ballFrame.add(ballPanel);
ballFrame.pack();
ballFrame.setVisible(true);
ballPanel.start();
```

This should produce an output similar to what is shown in *Figure 9.3*:

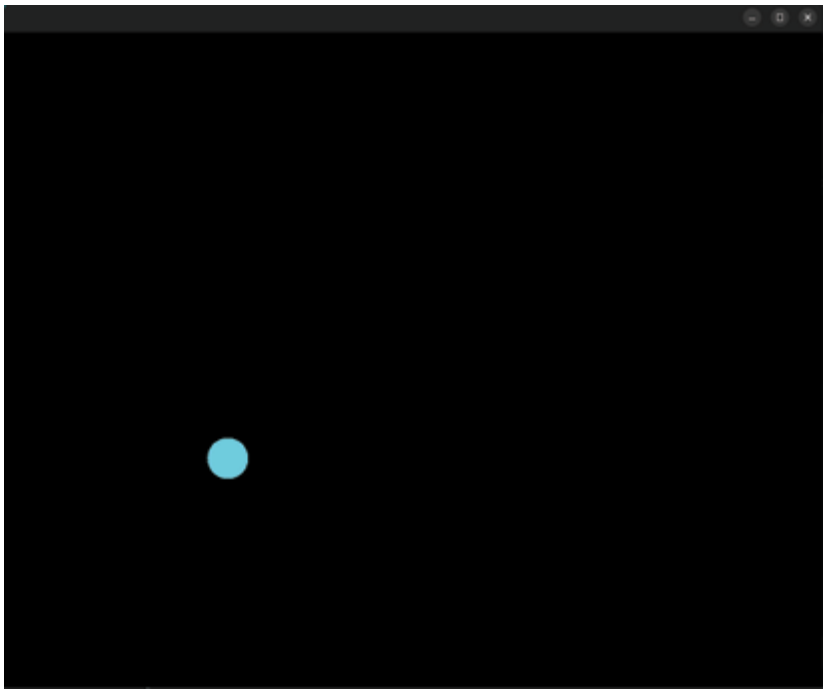

***Figure 9.3**: A window or panel showing a ball that moves across the screen, bouncing off the edges of the panel*

Algorithm

To solve this problem, we will take the following steps:

1. Define a new class named **Ball**. It should have:
    a. Imports for:
        i. **java.awt.Color**
        ii. **java.util.Random**

b. Class properties defined for:

i. Boolean (**movingUp**, **movingLeft**)

- Integer (**XCoord**, **YCoord**, **Size**, **Speed**)
- **Java.util.Color** (**Color**)

c. Getter and setter accessor methods defined for all properties.

d. A public constructor that:

i. Accepts parameters such as integer **size**, integer **speed**, integer **xCoord**, integer **yCoord**, and **Color color**.

- Defines a new **Random** object named **booleanDirection**.

ii. Sets **movingLeft** to the value of **booleanDirection**'s **nextBoolean()** method.

iii. Sets **movingUp** to the value of **booleanDirection**'s **nextBoolean()** method.

iv. Sets the **size**, **speed**, **xCoord**, **yCoord**, and **color** class properties to their respective, like-named parameters.

e. A public method named **update**. This method should:

i. Use an if statement to check the value of **movingLeft**. If it is **true**, then **xCoord** should be decremented by the value of **speed**. Otherwise (else), **xCoord** should be incremented by the value of **speed**.

ii. Use an if statement to check the value of **movingUp**. If it is **true**, then **yCoord** should be decremented by the value of **speed**. Otherwise (else), **yCoord** should be incremented by the value of **speed**.

2. Define a new class named **BallPanel** that extends the **JPanel** class, and implements the **Runnable** class. It should have:

a. Imports for:

i. **java.awt**(**Color**, **Dimension**, **Graphics**, **Graphics2D**)

- **java.util.Random**

ii. **javax.swing.JPanel**

b. The following constants defined:

i. A long named **serialVersionUID** set to a valid long value.

ii. An integer named **fPS** set to a value of **60**.

c. Class properties defined for:

i. Integers named **width** and **height**.

- **Ball** object named **ball**.

ii. **Thread** object named **panelThread**.

d. Getters and setters are not necessary for **BallPanel**'s class properties.

e. A public constructor that:

i. Accepts the integer parameters of **width** and **height**.

ii. Defines a new **Random** object named **randomInts**.

iii. Sets the **width** and **height** class properties to the values of their respective, like-named parameters.

iv. Invokes the **setPreferredSize()** method with a new **Dimension** object, while passing **width** and **height** as parameters, respectively.

v. Invokes the **setBackgroundColor()** method with **Color.BLACK** as a parameter.

vi. Invokes the **setFocusable()** method while passing **true** as a parameter.

vii. Instantiates the **panelThread** property to the result of **Thread.ofVirtual().unstarted(this)**.

viii. Instantiates the **ball** property as a new **Ball** while passing the values of **50**, **10**, **randomInts.nextInt(0,width)**, **randomInts.nextInt(0,height)**, and **Color.CYAN** to its constructor.

f. A public method named **run** that overrides the **run()** method from the **Runnable** class. Inside this method,

build a while loop that runs as long as **panelThread**'s **isAlive()** method returns a value of **true**. Inside the loop:

i. Call the **update()** method.

ii. Call the **repaint()** method.

iii. Implement a try/catch construct to trap for an **InterruptedExpection** named **e**, that calls the **Thread** class's **sleep()** method while passing the result of **1000 / fPS** as a parameter.

g. A public method named **start** that simply invokes **panelThread**'s **start()** method.

h. A public method named **paintComponent** that accepts a **Graphics** object named **g** as a parameter. Inside this method:

i. Call the **paintComponent()** method of the super class, while passing **g** as a parameter.

ii. Define a new **Graphics2D** object named **g2**, instantiated with the result of casting **g** as a **Graphics2D** object.

iii. Call **g2**'s **setColor()** method while passing the result of **ball**'s **getColor()** method as a parameter.

iv. Call **g2**'s **fillOval()** method while passing parameters such as **ball.getxCoord()**, **ball.getyCoord()**, **ball.getSize()**, and **ball.getSize()** (yes, **ball.getSize()** should be passed twice).

Call the **dispose()** method on the **g2** object.

i. A private method named **update**. This method will:

i. Call the **checkCollision()** method.

ii. Call ball's **update()** method.

j. A private method named **checkCollision**. This method will:

i. Use an if statement to check if the result of **ball**'s **getxCoord()** method is **<= 0** and if the result of

ball's **isMovingLeft()** is **true**. If true, call **ball**'s **setMovingLeft()** method with a parameter of **false**. Otherwise, check if the sum of **ball**'s **getxCoord()** + **getSize()** methods is >= **width** and if the result of **ball**'s **isMovingLeft()** is **false**. If true, call **ball**'s **setMovingLeft()** method with a parameter of **true**.

ii. Use another if statement to check if the result of **ball**'s **getyCoord()** method is <= **0** and if the result of **ball**'s **isMovingUp()** is **true**. If true, call **ball**'s **setMovingUp()** method with a parameter of **false**. Otherwise, check if the sum of **ball**'s **getyCoord()** + **getSize()** methods is >= **height** and if the result of **ball**'s **isMovingUp()** is **false**. If true, call ball's **setMovingUp()** method with a parameter of **true**.

Flowchart

The flowcharts for *Program 9.2*'s **Ball update()** and **BallPanel checkCollision()** methods can be seen in *Figures 9.4* and *9.5*, respectively:

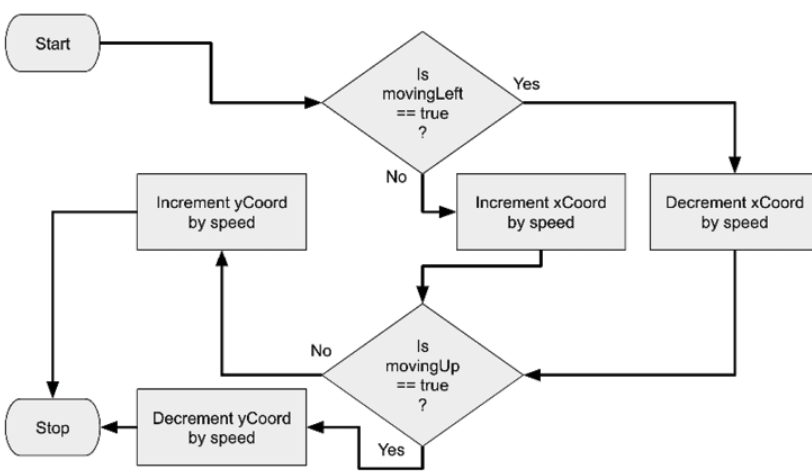

***Figure 9.4**: Flowchart describing the Ball class's update() method*

The following figure depicts the control logic that governs ball

movement and direction changes based on boundary checks and motion state:

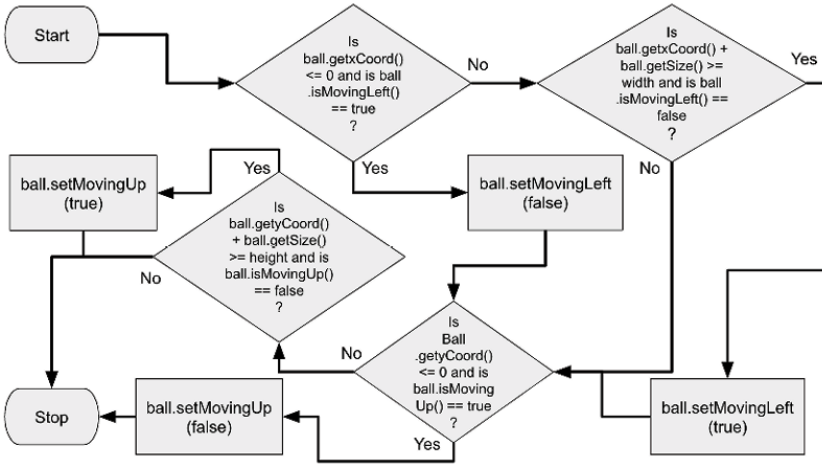

***Figure 9.5**: Flowchart describing the BallPanel class's checkCollision() method*

Java code

The Java code for *Program 9.2*'s **Ball** class is specified in the following section (getter and setter accessor methods omitted for brevity):

```
import java.awt.Color;
import java.util.Random;
public class Ball {
    private boolean movingUp;
    private boolean movingLeft;
    private int xCoord;
    private int yCoord;
    private int size;
    private int speed;
    private Color color;
    public Ball(int size, int speed, int xCoord, int yCoord, Color color)
{
```

```
        Random booleanDirection = new Random();
        movingLeft = booleanDirection.nextBoolean();
        movingUp = booleanDirection.nextBoolean();
        this.size = size;
        this.speed = speed;
        this.xCoord = xCoord;
        this.yCoord = yCoord;
        this.color = color;
    }
    public void update() {
        if (movingLeft) {
            xCoord -= speed;
        } else {
            xCoord += speed;
        }
        if (movingUp) {
            yCoord -= speed;
        } else {
            yCoord += speed;
        }
    }
}
```

The Java code for *Program 9.2*'s **BallPanel** class is specified in the following section (getter and setter accessor methods omitted for brevity):

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.Random;
```

```
import javax.swing.JPanel;
public class BallPanel extends JPanel implements Runnable {
    private static final long serialVersionUID =
2043779718874812497L;
    private final int fPS = 60; // frames per second
    private int width;
    private int height;
    private Ball ball;
    private Thread panelThread;
    public BallPanel(int width, int height) {
        Random randomInts = new Random();
        this.width = width;
        this.height = height;
        this.setPreferredSize(new Dimension(width, height));
        this.setBackground(Color.black);
        this.setFocusable(true);
        this.panelThread = Thread.ofVirtual().unstarted(this);
        ball = new Ball(50, 10,
                randomInts.nextInt(0,width),
                randomInts.nextInt(0,height),
                Color.CYAN);
    }
    @Override
    public void run() {
        while (panelThread.isAlive()) {
            update();
            repaint();
            try {
                Thread.sleep(1000 / fPS);
            } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }
}
public void start() {
    panelThread.start();
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    g2.setColor(ball.getColor());
    g2.fillOval(ball.getxCoord(), ball.getyCoord(),
            ball.getSize(), ball.getSize());
    g2.dispose();
}
private void update() {
    checkCollision();
    ball.update();
}
private void checkCollision() {
    if (ball.getxCoord() <= 0 && ball.isMovingLeft()) {
        ball.setMovingLeft(false);
    } else if (ball.getxCoord() + ball.getSize() >= width
            && !ball.isMovingLeft()) {
        ball.setMovingLeft(true);
    }
    if (ball.getyCoord() <= 0 && ball.isMovingUp()) {
        ball.setMovingUp(false);
    } else if (ball.getyCoord() + ball.getSize() >= height
            && !ball.isMovingUp()) {
```

```
            ball.setMovingUp(true);
        }
    }
}
```

Code explanation

To solve this problem, we will first build a new class named **Ball**. This class will not actually draw the ball, but it will help us maintain data about it and its movements. It will require two imports: **java.awt.Color** and **java.util.Random**. This class will also require several class properties: two Booleans named **movingUp** and **movingLeft**, four integers named **xCoord**, **yCoord**, **size**, and **speed**, and finally a **Color** object named **color**. All of these properties will require public getter and setter accessor methods.

The **Ball** class will also require a public constructor that accepts five parameters: Integers named **size**, **speed**, **xCoord**, and **yCoord**, and a **Color** object named **color**. Inside the constructor, we should create a new **Random** object named **booleanDirection**. The class properties **movingLeft** and **movingUp** should both be initialized with the result of **booleanDirection**'s **nextBoolean()** method. This will determine the initial direction of the ball. We should also assign the class properties of **size**, **speed**, **xCoord**, **yCoord**, and **color** to their respective, like-named parameters.

This class will also need a public method named **update**. The main task for this method is to update the ball's (**x,y**) coordinates based on its current speed and direction. It should first check if **movingLeft** is **true**, and if so, decrement **xCoord** by **speed**. If **movingLeft** is **false**, then the ball is assumed to be moving to the right, and **xCoord** should be incremented by **speed**. Additionally, this method should check if **movingUp** is **true**, and if so, decrement **yCoord** by **speed**. Otherwise, if **movingUp** is **false**, **yCoord** should be incremented by **speed**.

Note: The (x,y) coordinates of a JPanel begin with point (0,0) at the upper-left-most corner of the drawing surface, not the lower-left-most as seen on a normal Cartesian graph. In this way, a ball increasing its y coordinate will be moving down, and a ball decreasing its y coordinate towards zero will be moving up.

With the **Ball** class complete, we can begin work on a new class named **BallPanel**. This class will need six imports. Four from the **java.awt** library including the **Color**, **Dimension**, **Graphics**, and **Graphics2D** classes, **java.util.Random**, and **javax.swing.JPanel**. The **BallPanel** class should extend the **JPanel** class and implement the **Runnable** interface. As this is a serializable class, we will need to define a long constant named **serialVersionUID**. It is recommended to auto-generate the value of **serialVersionUID**.

This class will need a constant integer named **fPS** defined with a value of **60**. This constant will be the frames per second that we will use to determine our pause time between animation cycles. The **BallPanel** class will require four class properties: two integers named **height** and **width**, a **Ball** object named **ball**, and a **Thread** object named **panelThread**.

The **BallPanel** class will need a public constructor that accepts integer parameters named **width** and **height**. It should define a new **Random** object named **randomInts**. It should assign the class properties of **width** and **height** to the values of their like-named, respective parameters. Next, the constructor should define the size of the panel by calling the **setPreferredSize()** method while passing a new **Dimension** object constructed with the parameters of **width** and **height,** respectively. It should then set the background color of the panel by calling the **setBackground()** method while passing **Color.BLACK** as a parameter. It should also call the **setFocusable()** method while passing a value of **true** as a parameter.

The constructor should also instantiate the **panelThread** property to the result of the **Thread** class's **ofVirtual().unstarted(this)** method. This will utilize Java's virtual threading capabilities and allow us some fine-grained control over pause times.

Note: Virtual threading was implemented as a part of Project Loom in Java 21. The idea behind virtual threads, is that (Tyson 2023) they abstract the underlying operating system threads and are now managed by the JVM instead of the developer. This addition greatly simplified the usage of threads in Java.

Finally, this constructor should instantiate the **ball** property to a new **Ball**, while passing the following parameters to its constructor: **50**, **10**, **randomInts.nextInt(0,width)**, **randomInts.nextInt(0,width)**,

and **Color.CYAN**.

The **BallPanel** class will need a private method named **checkCollision**. The idea with this method is that once during each animation sequence, it will check to see if the ball has collided with one or more sides of the panel. If it has, its direction should be reversed.

First, we will check if the result of **ball**'s **getxCoord()** method is $<=$ to **0** and if the result of **ball**'s **isMovingLeft()** method is **true**. If so, then **ball** has *hit* the left side wall of the panel, so we will alter its direction by calling **ball**'s **setMovingLeft()** method while passing **false** as a parameter. If that is not the case, we should also check if the sum of **ball**'s **getxCoord()** method + **ball**'s **getSize()** method is $>=$ **width** and the result of **ball**'s **isMovingLeft()** method is **false**. In this case, the ball is moving right and has hit the right side wall of the panel, so we will alter its direction by calling **ball**'s **setMovingLeft()** method while passing **true** as a parameter.

Likewise, we will check if the result of **ball**'s **getyCoord()** method is $<=$ to **0** and if the result of **ball**'s **isMovingUp()** method is **true**. If so, then **ball** has *hi*t the top side wall of the panel, so we will alter its direction by calling **ball**'s **setMovingUp()** method while passing **false** as a parameter. If that is not the case, we should also check if the sum of **ball**'s **getyCoord()** method + **ball**'s **getSize()** method is $>=$ **height** and the result of **ball**'s **isMovingUp()** method is **false**. In this case, the ball is moving down and has hit the bottom side wall of the panel, so we will alter its direction by calling **ball**'s **setMovingUp()** method while passing **true** as a parameter.

The **BallPanel** class will need a private method named **update**. This method has two simple tasks. First, it calls the **checkCollision()** method to see if any directional changes are required. Finally, it calls **ball**'s **update()** method to move the ball to its new coordinates.

Implementing the **Runnable** class for threading will require us to override the public **run()** method with our own. Our **run()** method will implement a while loop to iterate as long as the result of **panelThread**'s **isAlive()** method returns a value of **true**. Inside the loop, we should call the **update()** method (to check for collisions and move the ball), and then we should call the **repaint()** method to force the panel to redraw itself (happens via the **paintComponent()**

method that we will write shortly). Additionally, we will invoke the **Thread** class's **sleep()** method while passing the quotient of **1000 / fPS** as a parameter. This should pause the thread's execution for 16⅔ milliseconds; enough time to update and repaint 60 times per second.

The **BallPanel** class will need a public method named **start**. This method will only need a single line of code, and that is to call **panelThread**'s **start()** method. This way, the ability to start **panelThread** is exposed in a public scope.

The **BallPanel** class will need a public method named **paintComponent**, that accepts a **Graphics** object named **g** as a parameter. We will start by calling the **paintComponent()** method of our super class (**JPanel**) while passing the **g** parameter. Next, we will define a **Graphics2D** object named **g2**, instantiated with the result of casting **g** to be a **Graphics2D** object. Next, we will set the drawing color by calling **g2**'s **setColor()** method while passing the result of **ball**'s **getColor()** method as a parameter. We will then draw the ball in its current position by calling **g2**'s **fillOval()** method while passing four parameters: **ball.getxCoord()**, **ball.getyCoord()**, **ball.getSize()**, and **ball.getSize()** a second time. Finally, we should call **g2**'s **dispose()** method.

With these two classes complete, our initial test code should now animate a cyan-colored ball bouncing around a window.

Program 9.3: Animate multiple bouncing balls

Draw multiple balls on a panel, start their movements in random directions, and have them *bounc*e off the edges of the panel and each other. We can test it with code similar to the following:

```
JFrame ballFrame = new JFrame();
ballFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
MultipleBallPanel ballPanel = new MultipleBallPanel(1000,800,20);
ballFrame.add(ballPanel);
ballFrame.pack();
ballFrame.setVisible(true);
```

```
ballPanel.start();
```

This should produce an output similar to what is shown in the following *Figure 9.6*:

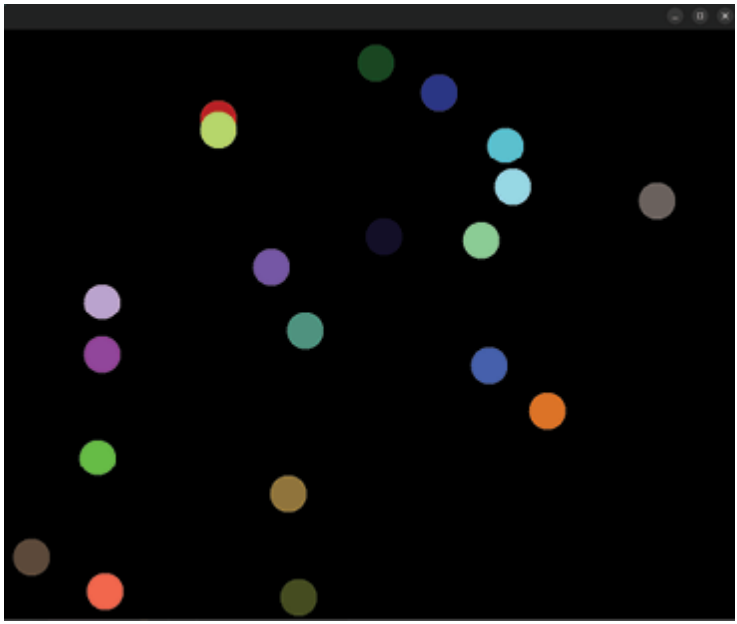

***Figure 9.6**: A window or panel showing 20 balls that move across the screen*

Algorithm

To solve this problem, we will take the following steps:

1. Similar to the steps for *Program 9.2*, we will need a class named **Ball**. Fortunately, we can reuse that same class without any additional modifications.
2. Define a new class named **MultipleBallPanel** that extends the **JPanel** class and implements the **Runnable** class. It should have:
    a. Imports for:
        i. **java.awt** (**Color**, **Dimension**, **Graphics**, **Graphics2D**)
        ı **Java.util** (**ArrayList**, **List**, **Random**)
        ı **javax.swing.JPanel**
    b. The following constants are defined:

i. A long-named **serialVersionUID** set to a valid long value.

ii. An integer named **fPS** set to a value of **60**.

c. Class properties defined for:

i. Integers named **width** and **height**.

ı **Thread** named **panelThread**.

ii. **List<Ball>** object named **ballList**.

iii. **Random** object named **randomInts**.

d. Getters and setters are not necessary for **BallPanel**'s class properties.

e. A public constructor that:

i. Accepts the integer parameters of **width**, **height**, and **numBalls**.

ii. Instantiates the **randomInts** property to a new **Random()**.

iii. Instantiates **ballList** to a new **ArrayList<>()**.

iv. Sets the **width** and **height** class properties to the values of their respective, like-named parameters.

v. Invokes the **setPreferredSize()** method with a new **Dimension** object, while passing **width** and **height** as parameters, respectively.

vi. Invokes the **setBackgroundColor()** method with **Color.BLACK** as a parameter.

vii. Invokes the **setFocusable()** method while passing **true** as a parameter.

viii. Instantiates the **panelThread** property to the result of **Thread.ofVirtual().unstarted(this)**.

ix. Build a for loop that defines a new integer named **ballCounter** with a value of 0, and iterates as long as **ballCounter** is less than **numBalls**.

x. Inside the for loop, add a new **Ball** to **ballList** while passing the values of **50**, **10**, **randomInts.nextInt(0,width)**,

randomInts.nextInt(0,height), and the result of the **randomColor()** method to the **Ball** constructor.

f. A public method named **run** that overrides the **run()** method from the **Runnable** class. Inside this method, build a while loop that runs as long as **panelThread**'s **isAlive()** method returns a value of **true**. Inside the loop:

   i. Call the **update()** method.

   ii. Call the **repaint()** method.

   iii. Implement a try/catch construct to trap for an **InterruptedExpection** named **e**, that calls the **Thread** class's **sleep()** method while passing the result of **1000 / fPS** as a parameter.

g. A public method named **start** that simply invokes **panelThread**'s **start()** method.

h. A public method named **paintComponent** that accepts a **Graphics** object named **g** as a parameter. Inside this method:

   i. Call the **paintComponent()** method of the super class, while passing **g** as a parameter.

   ii. Define a new **Graphics2D** object named **g2**, instantiated with the result of casting **g** as a **Graphics2D** object.

   iii. Define a for each loop to iterate through each **Ball** named **ball** in **ballList**.

   iv. Inside the loop, call **g2**'s **setColor()** method while passing the result of **ball**'s **getColor()** method as a parameter.

   v. Then, inside the preceding loop, call **g2**'s **fillOval()** method while passing **ball.getxCoord()**, **ball.getyCoord()**, **ball.getSize()**, and **ball.getSize()** parameters (yes, **ball.getSize()** should be passed twice).

   ı Call the **dispose()** method on the **g2** object.

i. A private method named **randomColor** that returns a **Color** object. This method will:

i. Define an integer named **red**, and set it to the result of **randomInts**'s **nextInt()** method while passing **0** and **256** as parameters.

ii. Define an integer named **green**, and set it to the result of **randomInts**'s **nextInt()** method while passing **0** and **256** as parameters.

iii. Define an integer named **blue**, and set it to the result of **randomInts**'s **nextInt()** method while passing **0** and **256** as parameters.

iv. Return a new **Color** to the calling program, while passing **red**, **green**, and **blue** as parameters to its constructor.

j. A private method named **update**. This method will define a for each loop to iterate through each **Ball** named **ball** in **ballList**. Inside the loop, call the **checkWallCollision(), checkBallCollision(),** and **ball**'s **update()** method.

k. A private method named **checkWallCollision** that accepts a **Ball** named **ball** as a parameter. This method will:

i. Use an if statement to check if the result of **ball**'s **getxCoord()** method is **<= 0** and if the result of **ball**'s **isMovingLeft()** is **true**. If true, call **ball**'s **setMovingLeft()** method with a parameter of **false**. Otherwise, check if the sum of **ball**'s **getxCoord()** + **getSize()** methods is **>= width** and if the result of **ball**'s **isMovingLeft()** is **false**. If true, call **ball**'s **setMovingLeft()** method with a parameter of **true**.

ii. Use another if statement to check if the result of **ball**'s **getyCoord()** method is **<= 0** and if the result of **ball**'s **isMovingUp()** is **true**. If true, call **ball**'s **setMovingUp()** method with a parameter of **false**. Otherwise, check if the sum of **ball**'s **getyCoord()** + **getSize()** methods is **>= height**

and if the result of **ball**'s **isMovingUp()** is **false**. If true, call **ball**'s **setMovingUp()** method with a parameter of **true**.

l. A private method named **checkBallCollision** that accepts a **Ball** named **ball** as a parameter. This method will define a for each loop to iterate through each **Ball** named **otherBall** in **ballList**.

   i. Inside the loop, use an if statement to check if **otherBall.getColor()** is **!=** **ball.getColor()**. If that statement is true, then build another if statement to check if the **ball.getxCoord()** **<=** **otherBall.getxCoord()** **+** **ball.getSize()**, **ball.getxCoord()** **>=** **otherBall.getxCoord()**, **ball.getyCoord()** **<=** **otherBall.getyCoord()** **+** **ball.getSize()**, **ball.getyCoord()** **>=** **otherBall.getyCoord()**.

   ii. If all of those conditions are true, then we have a ball-to-ball collision. Inside this statement, check if the result of **ball.isMovingLeft()** is **true**. If it is, call **ball.setMovingLeft(false)**. Otherwise, call **ball.setMovingLeft(true)**. Next, check if the result of **ballIsMovingUp()** is true. If it is, call **ball.setMovingUp(false)**. Otherwise, call **ball.setMovingUp(true)**.

Flowchart

The flowcharts for *Program 9.3* show **MultipleBallPanel**'s **paintComponent()** method in *Figure 9.7* and its **checkBallCollision()** method in *Figure 9.8*:

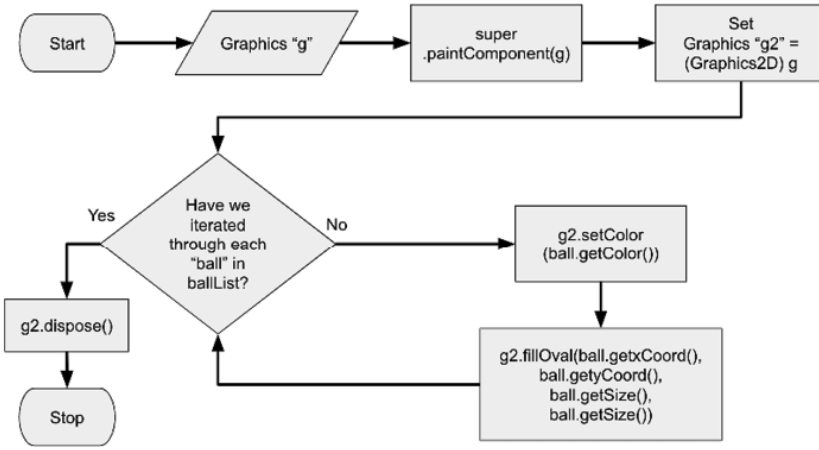

Figure 9.7: *Flowchart describing MultipleBallPanel's paintComponent() method*

The following figure illustrates the collision-detection and response logic used to evaluate interactions between the ball and other objects, updating movement direction accordingly:

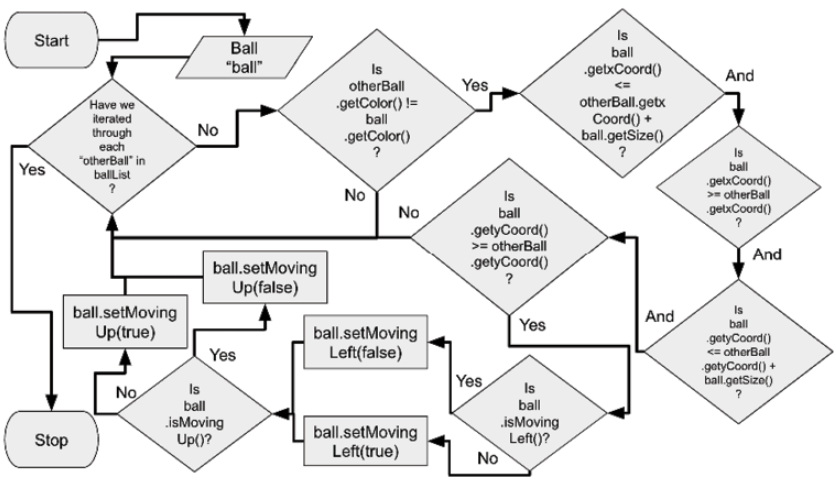

Figure 9.8: *Flowchart describing MultipleBallPanel's checkBallCollision() method*

Java code

The Java code for *Program 9.3*'s **MultipleBallPanel** class is quite lengthy, so we will break it up into a few sections. We will start with the imports as specified in the following section:

```
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import javax.swing.JPanel;
```

Next, we can start with the actual class. In the following section, we will instantiate several variables and our constructor:

```
public class MultipleBallPanel extends JPanel implements Runnable
{
   private static final long serialVersionUID =
2043779718874812497L;
   private final int fPS = 60; // frames per second
   private int width;
   private int height;
   private Thread panelThread;
   private List<Ball> ballList;
   private Random randomInts;
   public MultipleBallPanel(int width, int height, int numBalls) {
      randomInts = new Random();
      ballList = new ArrayList<>();
      this.width = width;
      this.height = height;
      this.setPreferredSize(new Dimension(width, height));
      this.setBackground(Color.black);
      this.setFocusable(true);
     this.panelThread = Thread.ofVirtual().unstarted(this);
     for (int ballCounter = 0; ballCounter < numBalls; ballCounter+
```

```
+) {
        ballList.add(new Ball(50, 10,
                randomInts.nextInt(0,width),
                randomInts.nextInt(0,height),
                randomColor()));
    }
}
```

Next, we will define our **run()** and **start()** methods to handle the virtual threads:

```
@Override
public void run() {
    while (panelThread.isAlive()) {
        update();
        repaint();
        try {
            Thread.sleep(1000 / fPS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
public void start() {
    panelThread.start();
}
```

Next, we will define our **paintComponent()** and **randomColor()** methods:

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D)g;
    for (Ball ball : ballList) {
```

```
            g2.setColor(ball.getColor());
            g2.fillOval(ball.getxCoord(), ball.getyCoord(),
                  ball.getSize(), ball.getSize());
        }
        g2.dispose();
    }
    private Color randomColor() {
        int red = randomInts.nextInt(0, 256);
        int green = randomInts.nextInt(0, 256);
        int blue = randomInts.nextInt(0, 256);
        return new Color(red,green,blue);
    }
```

Finally, we will define our **update()** method, as well as the **checkWallCollision()** and **checkBallCollision()** methods that it calls:

```
    private void update() {
        for (Ball ball : ballList) {
            checkWallCollision(ball);
            checkBallCollision(ball);
            ball.update();
        }
    }
    private void checkWallCollision(Ball ball) {
        if (ball.getxCoord() <= 0 && ball.isMovingLeft()) {
            ball.setMovingLeft(false);
        } else if (ball.getxCoord() + ball.getSize()
            >= width && !ball.isMovingLeft()) {
            ball.setMovingLeft(true);
        }
        if (ball.getyCoord() <= 0 && ball.isMovingUp()) {
            ball.setMovingUp(false);
```

```
        } else if (ball.getyCoord() + ball.getSize()
            >= height && !ball.isMovingUp()) {
            ball.setMovingUp(true);
        }
    }
    private void checkBallCollision(Ball ball) {
        for (Ball otherBall : ballList) {
            if (otherBall.getColor() != ball.getColor()) {
                if (ball.getxCoord() <= otherBall.getxCoord() +
ball.getSize()
                    && ball.getxCoord() >= otherBall.getxCoord()
                    && ball.getyCoord() <= otherBall.getyCoord() +
ball.getSize()
                    && ball.getyCoord() >= otherBall.getyCoord()) {
                    if (ball.isMovingLeft()) {
                        ball.setMovingLeft(false);
                    } else {
                        ball.setMovingLeft(true);
                    }
                    if (ball.isMovingUp()) {
                        ball.setMovingUp(false);
                    } else {
                        ball.setMovingUp(true);
                    }
                }
            }
        }
    }
}
```

Code explanation

So, the good news for solving this problem is that we do not have to start completely from scratch. While this program requires a little more depth and complexity in its **JPanel** code, we can reuse the **Ball** class that we built for *Program 9.2*.

We will, however, need a new class named **MultipleBallPanel** that extends **JPanel** and implements **Runnable**. This class will need eight imports. Four from the **java.awt** library, including the **Color**, **Dimension**, **Graphics**, and **Graphics2D** classes. Three from the **java.util** library including the **ArrayList**, **List**, and **Random** classes. Last, it will also need **javax.swing.JPanel**. As this is a serializable class, we will need to define a long constant named **serialVersionUID**. It is recommended to auto-generate the value of **serialVersionUID**.

This class will need a constant integer named **fPS** defined with a value of **60**. Similar to *Program 9.2*, this constant will be the frames per second that we will use to determine our pause time between animation cycles. The **MultipleBallPanel** class will require five class properties: two integers named **height** and **width**, a **Thread** object named **panelThread**, a **List<Ball>** named **ballList**, and a **Random** object named **randomInts**.

The **MultipleBallPanel** class will need a public constructor that accepts integer parameters named **width**, **height**, and **numBalls**. It should instantiate **randomInts** to a new **Random** object, and assign the class properties of **width** and **height** to the values of their like-named, respective parameters. Next, the constructor should define the size of the panel by calling the **setPreferredSize()** method while passing a new **Dimension** object constructed with the parameters of **width** and **height**, respectively. It should then set the background color of the panel by calling the **setBackground()** method while passing **Color.BLACK** as a parameter. It should also call the **setFocusable()** method while passing a value of **true** as a parameter.

The constructor should also instantiate the **panelThread** property to the result of the **Thread** class's **ofVirtual().unstarted(this)** method. We can finish the constructor by building a for loop to iterate a counter named **ballCounter** as long as **ballCounter** is < **numBalls**. Inside the loop, we will add a new **Ball** to **ballList**, while passing these parameters: **50**, **10**, **randomInts.nextInt(0,width)**, **randomInts.nextInt(0,height)**, and the result of a call to a new

randomColor() method.

As we just mentioned, the **MultipleBallPanel** class will need a private method named **randomColor** that returns an object of **Color**. This method is very straightforward as it only requires three randomly generated integers named **red**, **green**, and **blue**. Each integer is generated by **randomInts'** method **nextInt()** with parameters of **0** and **256**. This generates three random numbers ranging from 0 to 255. These values are then passed to a **Color** constructor to serve as individual values for the **red**, **green**, and **blue** components of a color. The new **Color** object is returned to the calling method. In this way, we can very easily randomly generate different colored balls.

The **MultipleBallPanel** class will need a private method named **checkWallCollision** that accepts a **Ball** named **ball**. This method will determine if a ball has collided with an edge or wall of the panel, and is identical to the **BallPanel**'s **checkCollision()** method that we described in *Program 9.2*. The only change here, is that it is named differently.

The **MultipleBallPanel** class will need a private method named **checkBallCollision** that accepts a **Ball** named **ball**. This method will check if a particular ball has collided with any of the other active balls on the panel. To that end, we will build a for each loop that iterates through each **Ball** in **ballList** named **otherBall**.

Inside the loop, we will check if **ball**'s color is not equal to **otherBall**'s color. This is how we will make sure that we are not checking **ball** for a collision against itself. In this way, a ball's color can do a sufficient job of doubling as a high cardinality identifier.

Note: Each RGB color is made up of three values ranging from 0 to 255, for a total of 16.77 million colors. The chances of two randomly generated colors matching exactly are extremely remote.

Assuming that the colors of **ball** and **otherBall** do not match, we then progress to our proximity check which consists of an if statement composed of four checks that must all be true for a collision to occur. Essentially, if the x and y coordinates of **ball** and **otherBall** come within the value returned by **ball.getSize()**, we have a collision. In that case, we will reverse **ball**'s direction by checking the values of its **movingUp** and **movingLeft** properties, and then

flipping their values.

The **MultipleBallPanel** class will need a private method named **update**. We will start this method by building a for each loop that iterates through each **Ball** in **ballList** named **ball**. Inside the loop, we will call our **checkWallCollision()** method while passing **ball** as a parameter. Next, we will similarly call our **checkBallCollision()** method while also passing **ball** as a parameter. As the last part of the loop, we will call **ball**'s **update()** method.

The **MultipleBallPanel** class will need a public method named **run**, that will override the **run()** method enforced by the **Runnable** interface. This **run()** method is identical to **BallPanel**'s **run()** method from *Program 9.2*.

Similarly, the **MultipleBallPanel** class will need a public method named **start**. This method only needs one line of code, and that is to call **panelThread**'s **start()** method.

Finally, the **MultipleBallPanel** class will need a public method named **paintComponent**, that accepts a **Graphics** object named **g**. Next, we will call the **paintComponent()** method of our super class (**JPanel**) while passing **g** as a parameter. Next, we will define a **Graphics2D** object named **g2**, instantiated with the result of casting **g** to be a **Graphics2D** object.

We will then implement a for each loop to iterate through each **Ball** in **ballList**, named **ball**. Inside the loop, we will set the drawing color to the result of **ball**'s **getColor()** method, and draw ball at its (x,y) coordinates and **size** property. The loop will continue until each ball has been drawn. Finally, we will call the **g2** object's **dispose()** method.

This should result in many balls moving on the panel, which bounce off of the walls and each other.

Conclusion

In this chapter, we discussed simple graphics and animation. What we found was that animating can be very complex and math-intensive. Especially when we have to compute whether our animated objects have collided with anything else, and also how they should react to such a collision.

Additionally, in this chapter, we learned about serializable classes and virtual threads. We also worked with super classes and interfaces.

Points to remember

Have a look at some key points to remember:

- Serializable classes are classes that Java can convert into byte streams. The serialVersionUIDs allow these classes to be accurately identified during the serialization process.
- A JPanel's (0,0) coordinates begin in the upper-left-most corner of the panel.
- It is considered a good idea to properly dispose of any Graphics objects that we have created ourselves.
- Virtual threads abstract away the details of operating system or platform threads, drastically increasing their ease of use.
- Our main programs all set their JFrames to exit on close. This helps with development, as we might close a panel/window and forget that the program is still running in the IDE.

Questions

Answer the following questions to check your understanding:

1. What do we call an oval with equal values for height and width?
2. Considering Program 9.1, how could we change it to randomly generate small, 10x10 pixel boxes instead of stars?
3. Considering Programs 9.2 and 9.3, we built the JPanel dimensions at 1000x800 pixels. Why would we not make the height and width equal in size?
4. Is Runnable a class or an interface? When writing a class that implements Runnable, which method should we override?
5. What would be another way that we could randomly generate a Color object?

6. Considering Program 9.3, how could we use a UUID as each ball's unique identifier?

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Appendix A

References

1. Baeldung (2024). Java Primitives Versus Objects. Retrieved on 2024-12-31 from: **https://www.baeldung.com/java-primitives-vs-objects**
2. IANA (2023). IANA IPv4 Address Space Registry. Retrieved on 2025-02-05 from: **https://www.iana.org/assignments/ipv4-address-space/ipv4-address-space.xml**
3. Nam T.V. (2024). Validating IPv4 Address in Java. Retrieved on 2025-02-05 from: **https://www.baeldung.com/java-validate-ipv4-address#use-regex**
4. Oracle (2024). RoundingMode (Java SE 23 and JDK 23). Retrieved on 2025-02-05 from: **https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/RoundingMode.html**
5. Ploetz A. (2024). Code with Java 21. BPB Publications. London, UK. (pp. 24)
6. Spring (2024). Spring Security. Retrieved on 2024-12-27 from: **https://docs.spring.io/spring-security/reference/features/authentication/password-**

storage.html#authentication-password-storage-bcrypt

7. Tyson (2023). Project Loom: Understand the new Java concurrency model. InfoWorld. Retrieved on 2025-03-08 from: **https://www.infoworld.com/article/2334607/project-loom-understand-the-new-java-concurrency-model.html**

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

APPENDIX B

Answers to Exercises

Chapter 1

1. *Program 1.1* could be rewritten with either a while loop or a traditional for loop. The disadvantages would be that there would be more code to write in both cases, and (in the case of the while loop) a counter would have to be incremented manually.
2. *Program 1.2* could not be rewritten with a for-each loop, because for-each loops cannot decrement their counters. Therefore, we would have to find another way to build the reversed String.
3. Technically, *Program 1.3* could be solved with two separate if conditions, shown as follows:

```
private static void fizzBuzz(int maxCount) {
    for (int counter = 1; counter < maxCount + 1; counter++) {
        if (counter % 3 == 0) {
            System.out.print("Fizz = ");
            System.out.println(counter);
```

```
        }
        if (counter % 5 == 0) {
            System.out.print("Buzz = ");
            System.out.println(counter);
        }
    System.out.println();
}
```

The output formatting would be a bit trickier, but technically this would indeed print both **Fizz** and **Buzz** if **counter** was evenly divisible by both 3 and 5, respectively.

4. We would have to check for single-digit numbers and print an additional space before each single-digit number. That would allow the output to be properly formatted.

5. Setting **number** to something like –1 might be a better solution. We could also set **number** to 0 right away and change the loop to a do-while loop. That would ensure it would run at least once, which is long enough for the input to change **number**.

6. To solve *Program 1.7* with a standard while loop, we could code it like this:

```
System.out.println("What is your name?");
String inputStr = inputScanner.nextLine();
while (!inputStr.equals("exit")) {
    System.out.printf("Hello %s!\n", inputStr);
    System.out.println("What is your name?");
    inputStr = inputScanner.nextLine();
}
```

The difference with this approach is that exit could be entered right away, and then the loop would never run.

Chapter 2

1. The **BufferedWriter** utilizes an internal buffer that helps to

abstract the disk writing process, and ultimately results in fewer disk operations, thus increasing performance.

2. When the **readLine()** method has finished reading a file, it will return a pointer to a value of null.
3. If a column contains a comma itself, that column will inadvertently be split into an additional column.
4. No, the values will be different because of the salt used by the **BCrypt** library.
5. It inherits from the **Exception** base class.
6. We could build the **currentUsers** list ahead of time and pass it as a parameter to the **writePasswordToFileUnique()** method.

Chapter 3

1. A for loop could be iterated backward by setting the counter variable to the max value, and decrementing the counter until the lowest value for the counter has been reached.
2. We could check if the ASCII code integer variable is greater than 96 and less than 123.
3. We could use the **String** class's **replace()** method, shown as follows:

```
someString.replace("\\","/");
```

4. As the 2600 model number appears in the string after the first space character, we could use the **String** class's **indexOf()** method to find it, and then set the variable with the result of the **substring()** method, demonstrated as follows:

```
String string = "ATARI 2600 CARTRIDGE GAMES";
int startIndex = string.indexOf(" ") + 1;
String model = string.substring(startIndex, startIndex + 4);
System.out.printf("The model is: %s\n", model);
```

5. The **length()** method will return the number of characters

present in a string.

6. A regular expression pattern to match a **String** of five lowercase characters might look like this:

```
Pattern.compile("[a-z]{5}");
```

7. The method to retrieve the value of the **game** property is as follows:

```
private static String getGameValue(String gameData) {
    String[] columns = gameData.split("[,]");
    int gameIndex = columns[1].indexOf(":") + 2;
    String game = columns[1].substring(gameIndex).trim();
    return game;
}
```

8. Instead of two separate if statements, we could join them together by using an else-if statement for the second condition. This way, it would only check for the location property if the team property was not present.

Chapter 4

1. *Program 4.1* could make more efficient use of its resources without the extra processing around defining the **isYNegative** variable, and then also checking whether **numY** is less than 0. After all, it checks this again further down in the program. Therefore, it would be more efficient to avoid defining, checking, and setting **isYNegative**, and to change the last if statement to the following:

```
if (numY < 0) {
    for (int counter = 0; counter > numY; counter--) {
        returnVal -= numX;
    }
} else {
```

2. An **addition** method can be built as follows:

```
public static int add(int numX, int numY) {
```

```
    int returnVal = numX;
    for (int counter = 0; counter < numY; counter++) {
        returnVal++;
    }
    return returnVal;
}
```

3. A **modulo** method could be built as follows:

```
private static int modulo(int dividend, int divisor) {
    int absoluteDividend = abs(dividend);
    int absoluteDivisor = abs(divisor);
    int total = 0;
    while ((total + absoluteDivisor) <= absoluteDividend) {
        total = total + absoluteDivisor;
    }
    return subtract(absoluteDividend, total);
}
```

4. The code for the new method would be as follows:

```
public static boolean isBitFlipped2(int number, int bit) {
    String num = Integer.toBinaryString(number);
    int length = num.length();
    for (int index = 0; index < length; index++) {
        if (Math.pow(2, index) == bit) {
            // if the index is equal to the bit we are looking for
            char currentChar = num.charAt(length - (index + 1));
            if (currentChar == '1') {
                return true; // bit is flipped
            } else {
                return false; // bit is not flipped
            }
        }
```

```
    }
    return false;
}
```

5. No, stringing the if statements together as one, large if-else would not work. This is because it is possible to send multiple error messages in a single code. If we used an if-else, then the logic would short-circuit and return to the calling program after identifying the first error.

6. There are a few different ways to accomplish this. One of the more efficient ways would be to add each error message to an **ArrayList**, and then use the **String join()** method to automatically convert it to a comma-delimited string, shown as follows:

```
private static String getErrorMessages(int errorNumber) {
    final int UNKNOWN_ERROR = 1;
    final int DIVIDE_BY_ZERO_ERROR = 2;
    final int INVALID_DATA_TYPE = 4;
    final int PARSING_ERROR = 8;
    List<String> returnVal = new ArrayList<>();
    if (isBitFlipped(errorNumber, PARSING_ERROR)) {
        returnVal.add("PARSING_ERROR");
    }
    if (isBitFlipped(errorNumber, INVALID_DATA_TYPE)) {
        returnVal.add("INVALID_DATA_TYPE");
    }
    if (isBitFlipped(errorNumber, DIVIDE_BY_ZERO_ERROR)) {
        returnVal.add("DIVIDE_BY_ZERO_ERROR");
    }
    if (isBitFlipped(errorNumber, UNKNOWN_ERROR)) {
        returnVal.add("UNKNOWN_ERROR");
    }
    return String.join(", ", returnVal);
```

```
}
```

Chapter 5

1. Currently, adding too many pallets via the **Stack** class will fail silently. At the very least, we should post an error message stating what happened.

An array to resize the stack might look something like this:

```
private void resizePalletsArray() {
    // increase the max number of items by 5
    maxCount = maxCount + 5;
    // instantiate larger array
    String[] newArray = new String[maxCount];
    // copy current pallets array to new
    for (int counter = 0; counter < stackCount; counter++) {
        newArray[counter] = pallets[counter];
    }
    // override current items array with new array
    pallets = newArray;
}
```

2. No, **palletCount** should not have a **setPalletCount()** method. The **palletCount** variable is managed privately inside of the **Stack** class. Exposing it publicly has the potential to introduce unforeseen behaviors (bugs) with the **Stack** class.

3. The code for a **getBack()** method could look something like this:

```
public String getBack() {
    if (currentCount > 0) {
        return aircraft[currentCount - 1];
    } else {
        return "Queue is empty";
    }
```

```
}
```

4. Code for a parameter-less constructor for the **Queue** class might look like this:

```
public Queue() {
    aircraft = new String[10];
}
```

Or:

```
public Queue() {
    this(10);
}
```

5. We can remove a **Step** from the linked list by using the following method:

```
public Step removeStep(UUID id) {
    Step returnVal = null;
    Step currentStep = firstStep;
    Step previousStep = null;
    boolean found = false;
    int counter = 0;
    while (counter < stepCount) {
        if (currentStep.getId().equals(id)) {
            returnVal = currentStep;
            found = true;
            break;
        }
        previousStep = currentStep;
        currentStep = currentStep.getNextStep();
    }
    if (found) {
        if (returnVal.getId().equals(firstStep.getId())) {
            firstStep = returnVal.getNextStep();
```

```
        } else {
            previousStep.setNextStep(returnVal.getNextStep());
        }
    }
    return returnVal;
}
```

Chapter 6

1. We are using the **matches()** method because it matches the pattern against the entire string, not just a subset (substring) of it.
2. The dollar sign (**$**) is the metacharacter used with regular expressions to indicate the end of a string.
3. At the end of our regular expression pattern, we can enforce a minimum and a maximum number of characters by adding a dot (**.**) followed by the minimum and maximum specified as a pair of comma-delimited integers, encapsulated in braces (**{}**). For instance, we could specify that a matching string has a minimum of 10 characters and a maximum of 25 characters by adding this to the end of the pattern:

```
.{10,25}
```

4. The primary benefit of the **HALF_EVEN** rounding mode is that it statistically reduces the margin of error that can occur over multiple, cumulative calculations.
5. There may be a few ways to approach this problem, but this is one such solution:

```
Pattern platePattern = Pattern.compile("^[A-Z]{3}[-]{1}[0-9]{3}$");
```

6. A switch/case statement to see if an **Object number** is of an **Integer**, **Double**, or **Float** can be written as follows:

```
private static String whichKindOfNumber(Object number) {
    switch (number) {
        case Integer i:
```

```
            return "Integer";
        case Float f:
            return "Float";
        case Double d:
            return "Double";
        default:
            return "Not an Integer, Float, or Double";
    }
}
```

Chapter 7

1. The problem that we would have to solve here is that the **addHistoryItem()** method would continue to work just fine, but the **historyItemCount** would be incorrect (too high, due to the orphaned items). We could solve that by building a new method in our **TwoWayLinkedList** class to count each item in the linked list, shown as follows:

```
private int countHistoryItemsToCurrent(HistoryItem current) {
    int itemCount = 1;
    HistoryItem pointer = firstHistoryItem;
    while (pointer != null) {
        itemCount++;
        pointer = pointer.getNextHistoryItem();
        if (pointer == current) {
            break;
        }
    }
    return itemCount;
}
```

   Next, we could need to adjust how **historyItemCount** is assigned its value inside the **addHistoryItem()** method. Essentially, if the linked list is not empty, we call the

countHistoryItemsToCurrent() method. If it is empty, we simply set **historyItemCount** to **1**, demonstrated as follows:

```
public void addHistoryItem(HistoryItem histItem) {
    if (historyItemCount > 0) {
        HistoryItem currentLast = lastHistoryItem;
        currentLast.setNextHistoryItem(histItem);
        histItem.setPreviousHistoryItem(currentLast);
        historyItemCount = countHistoryItemsToCurrent(histItem);
    } else {
        firstHistoryItem = histItem;
        historyItemCount = 1;
    }
    lastHistoryItem = histItem;
}
```

2. The problem to solve here is that the deleted node's children should still be valid entries in the binary tree. To get this to work properly, we will first need to define getter/setter accessor methods for the **Node** class's **leftChild** and **rightChild** property. We would also need to add a **parent** property (of type **Node**) with its own accessor methods.

   Once we have done that, we could build a **deleteByKey()** method in our **BinaryTree** class like this:

```
public void deleteByKey(int key) {
    Node nodeToDelete = find(key);
    // save off the parent and children of the node to delete
    Node leftChild = nodeToDelete.getLeftChild();
    Node rightChild = nodeToDelete.getRightChild();
    Node parent = nodeToDelete.getParent();
    // delete the node
    if (parent.getLeftChild() == nodeToDelete) {
        parent.setLeftChild(null);
    } else {
```

```
        parent.setRightChild(null);
    }
    // re-add the deleted node's children to the binary tree
    if (leftChild != null) {
        insert(leftChild);
    }
    if (rightChild != null) {
        insert(rightChild);
    }
}
```

3. A constructor that builds the **Tree** from an array of **Nodes** can look like this:

```
public BinaryTree(Node[] nodes) {
    for (Node node : nodes) {
        insert(node);
    }
}
```

4. We could reverse the sign of the return values in our **PatientComparator**'s **compare()** method. That would treat the higher number as a higher priority and keep the largest value at the top of the heap, essentially converting our priority queue into a max heap:

```
@Override
public int compare(Patient p1, Patient p2) {
    if (p1.getPriority() > p2.getPriority()) {
        return -1;
    } else if (p1.getPriority() < p2.getPriority()) {
        return 1;
    } else {
        return 0;
    }
```

```
}
```

5. A parameter-less constructor for the class would be pretty simple, and would just have to initialize **patientHeap** as a new **PriorityQueue**:

```
public MedicalHeapJava() {
    patientHeap = new PriorityQueue<>(new PatientComparator());
}
```

To populate the heap, we could simply iterate through the array of patients and call the **push()** method to add each of them.

Chapter 8

1. The **String compareTo()** method compares the strings *lexicographically*, essentially decoding each String into tokens of the ASCII or UTF codes for each character. If the first array is determined to be greater than the second array, then the **compareTo()** method returns the positive difference. Otherwise, it returns the negative difference.

2. If we have even flipped just one pair, it is possible that we may still have items that are out-of-order. Keeping track of that lets us know if we need to rerun the sort.

3. While building our own method to split an array was a good exercise, the **java.util.Arrays** class has a static method called **copyOfRange()** that can also do this. For example:

```
leftArray = Arrays.copyOfRange(array, 0, middle);
```

4. At the end of the merge sort process, there should only be items remaining in either the **leftArray** or the **rightArray**, but not both. Therefore, it does not really matter which array is processed first.

5. This can be done with the **addAll()** method.

6. A constructor for the **BinaryTree** class that accepts a **List** of **Player** as a parameter and inserts them is as follows:

```
public BinaryTree(List<Player> players) {
    for (Player player : players) {
```

```
        insert(player);
    }
}
```

Chapter 9

1. An oval drawn with a height and width that are the same will appear to be a *circle*.
2. Instead of using the **Graphics2D** class's **drawLine()** method, we could use its **drawRect()** method to draw a *box*, shown as follows:

```
g2.fillR4ct(x, y, x + 10, y + 10);
```

3. The balls move at 45-degree angles. If the panels are of equal dimensions, then each ball will always take the same diamond-like path and never deviate from it. Building the panel with dimensions that are not equal will allow each ball to take different paths.
4. The **Runnable** class is an interface. In fact, it is a special kind of interface known as a **functional interface** that has only one abstract method. In this case, that method is the **run()** method, which we should override.
5. Instead of randomly generating three values from 0 to 255, we could always just randomly generate a single value from 0 to 16,777,216, demonstrated as follows:

```
int color = randomInts.nextInt(0, 16777216);
return new Color(color);
```

6. We could define a class property of the **Universally Unique Identifier** (**UUID**) type named *id* in the **Ball** class, and have it populated with a randomly generated UUID in the constructor. We could then use this identifier to check for collisions between each ball. This would offer a more unique identifier than a ball's color.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech

happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com

Index

A

Array Print 4
Array Print, concepts
  Algorithm, ensuring 4
  Code, implementing 5
  Flowchart 5
  Illustrating 5
Array Reverse 6
Array Reverse, concepts
  Algorithm, ensuring 6
  Code, implementing 7
  Flowchart 6
  Illustrating 7

B

Binary Error Handler 86
Binary Error Handler, concepts
  Algorithm, ensuring 86
  Code, implementing 88
  Flowchart 86
  Illustrating 89-91
Binary Tree 145
Binary Tree, concepts
  Algorithm, ensuring 146, 147
  Code, implementing 149, 150
  Flowchart 148
  Illustrating 151-153
Binary Tree Sort 177
Binary Tree Sort, concepts
  Algorithm, ensuring 177-179
  Code, implementing 181-183
  Flowchart 179, 180
  Illustrating 184-186
BooleanDirection 195
BooleanDirection, concepts
  Algorithm, ensuring 196-198
  Code, implementing 199, 201

Flowchart 198, 199
Illustrating 203-205
Bubble Sort 165
Bubble Sort, concepts
Algorithm, ensuring 166
Code, implementing 167
Flowchart 167
Illustrating 168, 169

C

Comma-Separated Value (CSV) File 30
CSV File, concepts
Algorithm, ensuring 31
Code, implementing 32
Flowchart 32
Illustrating 33, 34

D

Diamond Outline 13
Diamond Outline, concepts
Algorithm, ensuring 14
Code, implementing 15
Flowchart 15
Illustrating 17
Diamond Shape Number 10
Diamond Shape Number, concepts
Algorithm, ensuring 10
Code, implementing 11
Flowchart 11
Illustrating 12, 13
Divide Numbers 79
Divide Numbers, concepts
Algorithm, ensuring 79
Code, implementing 81
Flowchart 80
Illustrating 82

E

Even Number 82
Even Number, concepts
Algorithm, ensuring 83
Code, implementing 84
Flowchart 83
Illustrating 85

F

FizzBuzz 7
FizzBuzz, concepts
  Algorithm, ensuring 8
  Code, implementing 9
  Flowchart 8
  Illustrating 9

H

HistoryItem 136
HistoryItem, concepts
  Algorithm, ensuring 137-139
  Code, implementing 141-143
  Flowchart 139, 140
  Illustrating 144, 145

K

KvPair 64
KvPair, concepts
  Algorithm, ensuring 65
  Code, implementing 66
  Flowchart 66
  Illustrating 67, 68

L

Linked List 104
Linked List, concepts
  Algorithm, ensuring 106
  Code, implementing 107
  Flowchart 106
  Illustrating 109, 110

M

Merge Sort 170
Merge Sort, concepts
  Algorithm, ensuring 170-172
  Code, implementing 174
  Flowchart 172, 173
  Illustrating 175, 176
MotorVehicle 124
MotorVehicle, concepts
  Algorithm, ensuring 125, 126
  Code, implementing 127
  Flowchart 126
  Illustrating 129-131

MultipleBallPanel 205
MultipleBallPanel, concepts
  Algorithm, ensuring 206-208
  Code, implementing 210-213
  Flowchart 209
  Illustrating 214-216
Multiply Numbers 74
Multiply Numbers, concepts
  Algorithm, ensuring 74
  Code, implementing 75
  Flowchart 75
  Illustrating 76

P

Palindrome 62
Palindrome, concepts
  Algorithm, ensuring 62
  Code, implementing 63
  Flowchart 63
  Illustrating 64
Password File 34
Password File, concepts
  Algorithm, ensuring 35
  Code, implementing 36
  Flowchart 35
  Illustrating 36, 37
Password File Unique 42
Password File Unique, concepts
  Algorithm, ensuring 43
  Code, implementing 44
  Flowchart 44
  Illustrating 46, 47
Phone Number 55
Phone Number, concepts
  Algorithm, ensuring 55
  Code, implementing 56
  Flowchart 56
  Illustrating 57
Primitve Uppercase 52
Primitve Uppercase, concepts
  Algorithm, ensuring 52
  Code, implementing 53
  Flowchart 52
  Illustrating 53, 54
PriorityQueue 153
PriorityQueue, concepts
  Algorithm, ensuring 155, 156

Code, implementing 158, 159
Flowchart 156, 157
Illustrating 161, 162

Q

Queue 99
Queue, concepts
Algorithm, ensuring 100
Code, implementing 102
Flowchart 101
Illustrating 103, 104

R

Read Text File 28
Read Text File, concepts
Algorithm, ensuring 28
Code, implementing 29
Flowchart 29
Illustrating 30
Reference Uppercase 54
Reference Uppercase, concepts
Algorithm, ensuring 54
Code, implementing 55
Flowchart 54
Illustrating 55
Reverse String 50
Reverse String, concepts
Algorithm, ensuring 50
Code, implementing 51
Flowchart 51
Illustrating 52

S

Stack 94
Stack, concepts
Algorithm, ensuring 95
Code, implementing 97
Flowchart 96
Illustrating 98, 99
StarFieldPanel 190
StarFieldPanel, concepts
Algorithm, ensuring 190, 191
Code, implementing 192
Flowchart 192
Illustrating 194, 195
State Code Regular Expression 60

State Code Regular Expression, concepts
Algorithm, ensuring 60
Code, implementing 61
Flowchart 60
Illustrating 61, 62
State Code Valid 58
State Code Valid, concepts
Algorithm, ensuring 58
Code, implementing 59
Flowchart 58
Illustrating 59, 60
Subtract Numbers 77
Subtract Numbers, concepts
Algorithm, ensuring 77
Code, implementing 78
Flowchart 77
Illustrating 78, 79

T

Text File 25
Text File, concepts
Algorithm, ensuring 26
Code, implementing 27
Flowchart 26
Illustrating 27, 28

U

User Name 21
User Name, concepts
Algorithm, ensuring 21
Code, implementing 22
Flowchart 22
Illustrating 23

V

Validate IP Address 120
Validate IP Address, concepts
Algorithm, ensuring 122
Code, implementing 122
Flowchart 122
Illustrating 123, 124
Validate Password 117
Validate Password, concepts
Algorithm, ensuring 118
Code, implementing 119
Flowchart 119

Illustrating 120
Validate Password, configuring 118
Validate User File 38
Validate User File, concepts
Algorithm, ensuring 38
Code, implementing 39
Flowchart 39
Illustrating 40-42
Valid Number 18
Valid Number, concepts
Algorithm, ensuring 18
Code, implementing 19
Flowchart 19
Illustrating 20, 21
Valid Phone Number 114
Valid Phone Number, concepts
Algorithm, ensuring 115
Code, implementing 116
Flowchart 115
Illustrating 116, 117